Universidade do Minho
Escola de Engenharia

Tiago Paulo de Sousa Barros

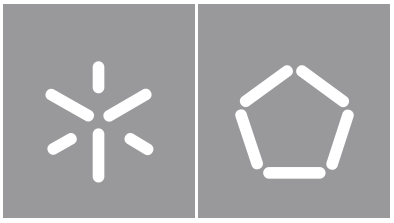
Remote Boot Manager: Operating System Installer

December 2019

Remote Boot Manager: Operating System Installer

Tiago Barros

UMinho | 2019



Universidade do Minho
Escola de Engenharia

Tiago Paulo de Sousa Barros

Remote Boot Manager: Operating System Installer

Dissertação de Mestrado
Engenharia Eletrónica Industrial e Computadores

Trabalho efetuado sob a orientação do
Professor Doutor Jorge Cabral

December 2019

Acknowledgements

I would like to thank Professor Jorge Cabral for all the guidance, trust and presence.

A big thank to my mom and dad, who always backed me up and supported me, making me never giving up until i fulfill my dreams and ambitions.

To everyone in ESRG for all the knowledge passed through this years and experience, thank you.

To everyone in my family for helping me in every situation i needed and giving me confidence and appreciation.

To my girlfriend Ana for giving me the love, friendship, support and for helping me in every decision of my life.

To all my friends, who stayed always with me in every situation and helped during this exceptional five years, you will always be in my mind.

To my friends and professors from Erasmus, who made those six months exceptional and made me grow up and learn much more, in special to Alex and professor Sérgio Montenegro, for receiving us and helping us so well.

Thank you every one, for making this five years the best of my life.

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

Resumo

Em 2018, o numero de Raspberry Pi vendidas passou os 23 milhões , mostrando a importância deste dispositivo na IoT e como está a afetar o mundo. Conectando cada dispositivo, é possível criar um cluster, um conjunto de computadores ligados que trabalha como um sistema único, aumentado o poder de processamento com um baixo consumo de energia.

Apesar de funcionar como um sistema único, um cluster envolve vários dispositivos, que precisam de ser reiniciados e configurados. Quando feitas uma por uma, estas tarefas consomem muito tempo e podem ser bastante monótonas.

Usando TCP/IP e FTP, um instalador de sistemas operativos remoto foi desenvolvido de forma a ser controlado remotamente por uma Aplicação Central. TCP/IP é usado para a comunicação entre os vários pontos, enquanto que FTP é usado para transferir imagens. A aplicação descarrega e instala o sistema operativo comandado pela Aplicação Central ao mesmo tempo que informa acerca do estado da instalação.

À medida que a tecnologia avança, as pessoas procuram formas para detetar erros e funcionamentos defeituosos o mais rapidamente possível pois gera perdas económicas, e desperdícios de tempo. Sistemas tolerantes a falhas são bastante convenientes nestas situações pois podem detetar e corrigir esses maus funcionamentos.

Para ultrapassar estes problemas, foi incluído no sistema um modo de reiniciar a Raspberry Pi em caso de mau funcionamento. Usando o Linux Watchdog, é possível ultrapassar este problema sem o uso de hardware adicional.

Quando um novo sistema operativo é instalado, também é preciso ser monitorizado, portanto um daemon foi desenvolvido para assegurar que o dispositivo consegue ainda comunicar com a Aplicação Central.

O grande desafio desta Tese de Mestrado é juntar a capacidade de, monitorizar e gerir sistemas operativos, comunicar e controlar Raspberry Pi's e evitar maus funcionamentos de sistemas operativos numa única ferramenta, enquanto mantendo amigo do utilizador e acessível para todos os que quiserem usar.

Palavras-chave: FTP, Instalador de Sistemas Operativos, Raspberry Pi, TCP/IP, Tolerância de Falhas.

Abstract

In 2018, the number of sold Raspberry Pi computers surpassed the 23 millions , showing the importance of this device in IoT and how it is affecting the world.

By connecting each device, is possible to create a cluster, a set of connected computers that works together as a single system, increasing the processing power with a low power consumption.

Despite acting like a single system, a cluster involves many devices, which need to be individually rebooted and configured. When done one by one, this tasks consume a lot of time and can be very monotonous.

Using TCP/IP and FTP, a remote operating system installer was developed so that it can be remotely controlled by a Central Application. TCP/IP is used for the communication between both endpoints, while FTP is used to transfer the images. The application downloads and installs the operating system commanded by the Central Application while informing about the installation status.

As technologies advance, people look for ways to detect errors and malfunctions as quickly as possible because it generates economical losses, and wastes time. Fault tolerance systems come very handy in these situations because they can detect and override these malfunctions.

To overcome this problems, it was included in the system a way to reboot the Raspberry Pi in case of malfunctions. Using the Linux Watchdog, is possible to overcome this problem without the usage of external hardware.

When a new operating system is installed, it also needs to be monitored, so a daemon was developed so that it can assure that the device can still communicate with the Central Application.

The big challenge of this Master Thesis is to join the capability of, monitoring and managing operating systems, communicate and control Raspberry Pi's and avoid operating systems malfunctions in a single tool, while also making it user friendly and available to everyone who wants to use it.

Keywords: Fault Tolerance, FTP, Operating System Installer, Raspberry Pi, TCP/IP.

Table of Contents

List of Figures

List of Tables

Acronyms

ACK Acknowledge.

App Application.

CPU Central Processing Unit.

DHCP Dynamic Host Configuration Protocol.

DOS Disk Operating System.

FTP File Transfer Protocol.

FTPS File Transfer Protocol over SSL.

GUI Graphical User Interface.

HTTP Hypertext Transfer Protocol.

IP Internet Protocol Access Control.

JSON Javascript Object Notation.

MAC Media Access Control.

OS Operating System.

gpu Graphics Processing Unit.

PXE Preboot Execution Environment.

wlan Wireless Local Area Network.

RBM Rewmote Boot Manager.

ROM Read-only Memory.

RPI Raspberry Pi.

SCTP Stream Control Transmission Protocol Execution Environment.

SD Secure Digital.

SFTP Simple File Transfer Protocol.

SoC System on Chip.

SSH Secure Shell.

SSL Secure Sockets Layer.

TCP Transfer Communication Protocol.

TLS Transport Layer Security.

UDP User Datagram Protocol.

VNC Virtual Network Computing.

WAN Wide Area Network.

Chapter 1

Introduction

Detecting and solving errors and malfunctions is a major concern nowadays,specially at operating system and kernel level where they are common, because they generate wastes of money and time that can be critical for projects [1].

When working with clusters, the number of devices to be monitored increases drastically. Each device needs to be configured individually and periodically rebooted which is a time wasting task. By developing a tool capable of sending commands to multiple devices and control the installed operating systems while monitoring them could save a lot of precious time and resources.

The main goal of this Master's Dissertation, is to provide a way to monitor and control Raspberry Pi's remotely using software. This is going to be achieved by developing an application capable of installing operating systems and perform commands remotely.

After an operating system is installed, a way of keeping the communication and monitor it will also be developed using daemons.

To avoid malfunctions in the system, a watchdog that resets the system in case of need is also going to be used.

1.1 Contextualization

This dissertation was developed as a project from University of Minho Embedded System Research Group Lab. Many projects that have been or that are still being developed include the usage of IoT devices like the Raspberry Pi.

To help monitor and manage the operating systems of these devices, an application with these features was necessary. It can be applied for small projects where only one Raspberry Pi is being used, as well as bigger projects with multiple devices like clusters.

This application relieves the users from the task of having to monitor periodically their devices, updating multiple operating systems, changing configurations of multiple devices and having to periodically manual reboot them.

In case of projects involving clusters, usually configurations and reboots need to be made to multiple devices which is a critical step in an operating system. If any error occurs, the user needs to be present near the device to solve it. With this application and the fault tolerance tools, these cases could be avoided by detecting in the Central App(Application) the error, and solved with the watchdog automatically rebooting the system, saving precious time for the development of projects.

In more critical cases like collision detection systems, the used devices need to be working at all the time. When a device malfunctions, it can only be detected after some time, which in some cases can be crucial due to the fact that any accident can happen where that device should be working. Not only this raises concerns about this problem as well it is causing the company that developed these systems to lose money because someone has to be called in person to the place where the device stopped malfunctioning, which in many cases involves big distances.

1.2 Motivation

Everyday time constraints are crucial for projects development. So having tools that help save time and that eliminates monotony are very important, either for companies as well for personal uses.

With the increase of IoT devices and clusters spread over the world [2] the number of devices that need to be monitored is huge [3], and keeps increasing over the years [4].

Raspberry Pi's can be used for a wide range of applications, since games, security or even data processing in case of the clusters [5].

With the research for options to fulfil the needs for Embedded System Research Group Lab, was detected that none could fill them all. With this, the project for the development of this dissertation started.

The main goal of this dissertation is to provide a tool that can be used to monitor,control and install operating systems on Raspberry Pi devices. The application needs to be user friendly and easy to use, to allow being used in all kind of projects that require it.

1.3 Objectives

In order to achieve the goal specified for this dissertation, the following objectives were set:

- Set constraints and requirements;
- Design the system taking into consideration the global overview;
- Divide the system into different modules
- Design all GUI(Graphical User Interface);
- Implement TCP/IP communication;
- Install different operating systems;
- Generate a small image for the application;
- Use threads to avoid GUI malfunctions;
- Implement a communication language;
- Implement FTP listing and file transferring;
- Assign functions to the implemented GUI;

1.4 Dissertation Structure

This document is split into six chapters that follow the development process and the necessary steps to design and develop this dissertation.

The first chapter gives to the reader an introduction to the contents of this document, its objectives and motivations, while also giving a general idea of what will be presented and explained

The second chapter contains a research of the available technologies and products in the market, as well as some theoretical concepts necessary to understand what will be explained in this dissertation. With it, is possible to contrast what is present in the market and what was developed highlighting the pros and cons of each solution. This research is heavily focused on bootloaders, operating system installers and communication protocols.

The third chapter gives an overview of how the system was designed and the choices made regarding the research made in the second chapter.

The fourth chapter shows to the user how the system was developed taking into account the choices made in the third chapter. This chapter is divided into three subsections that match the three developed components.

The fifth chapter shows the results obtained with the development of this dissertation as well as the tests made to prove that everything works as intended.

The final chapter gives a conclusion of what was developed and written in this dissertation and what can be improved and implemented in the future to improve it.

With this structure is possible to verify that every step is necessary to advance to the next one, because they are dependent of each one.

Chapter 2

State of the Art

2.1 Introduction

To better understand the market of RPI(Raspberry Pi) bootloaders and operating system installers it is necessary to search what options are currently available and being distributed.

One of the main objectives of this dissertation is the possibility of remote controlling the OS(Operating System) installation using a Desktop application, so one part of the research was the existence of boot managers with this feature.

The final objective of this dissertation is to use the Raspberry Pi to install the desired operating system, maintaining the manager intact yet, part of the research included some of the usual Desktop operating system installers to understand how they operate in contrast to the Raspberry Pi installers.

Another objective of this dissertation is the fault manager daemon with TCP/IP communication, to avoid the Raspberry Pi blackout and to keep the communication with the Desktop application. So it is necessary to search for current solutions for this problem and the missing requirements.

2.2 NOOBS - New Out Of Box Software

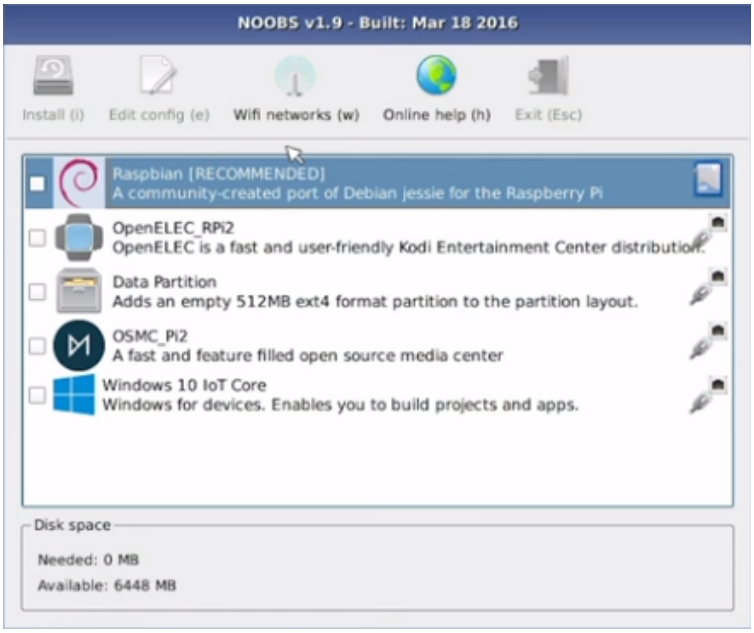


Figure 2.1: NOOBS Graphical Interface.

NOOBS (New Out Of Box Software) is an Operating System Installer released on 3rd July 2013 that helps to select and install systems for the Raspberry Pi [6]. It is an official Raspberry Pi Foundation software that was made to abstract users from the steps to manually install an operating system on the Raspberry Pi [7].

This software supports the installation of Arch Linux ARM, OpenELEC, Pidora, RaspBMC, Raspbian and RiscOS and more custom operating systems [8]. All these operating systems can also be found in the Raspberry PI official website.

NOOBS runs on a buildroot minimal operating system which has all the necessary drivers and packages needed which are used by the graphical application which was developed using QT. This application gives the user the possibility to download and install images from the internet or to install customized images present on the SD(Secure Digital) card in a easy way.

On first boot,NOOBS creates a partition table with the first one being the recovery partition which contains the recovery application, where the operating systems can be selected and installed. The second partition is an extended partition which contains the "Settings" [9]. The Settings partition is where the user can edit some options of the recovery partition without booting the Raspberry Pi. This makes easier to customize the recovery partition and gives the possibility to automate some aspects of the recovery application.

Despite allowing dual boot, a big problem of NOOBS is that if the user wants to change any operating system or install a new one, all the operating systems that are currently installed need to be erased and reinstalled.

A version of NOOBS containing all the available images is also given for download in the Raspberry Foundation webpage. To make it even easier to set-up the Raspberry Pi for beginners, Raspberry Pi Foundation sells SD cards with NOOBS already installed and configured [10].

2.3 PINN - PINN is not NOOBS

As the name says, PINN is not Noobs but is a fork of him,with its own developing team and also open source, offering a larger selection of operating systems, added features and fixed bugs [11]. This operating system installer was published by Matt Huisman in 2013 and is still an open-source project.

In total, 64 operating systems can be installed [12] if space is available and multiple operating systems can be installed in dual boot.

The main features of PINN are:

- Multi Operating System (OS) installer
- Operating system selector
- Boot manager
- Run low-level utilities
- OS maintenance utility:

    Backup and Restore OSes

    Recovery shell

    SD card clone utility

    Password restorer

    File System Checker

A very useful feature when compared to NOOBS, is the possibility to install operating systems on external storage instead of the SD card, and to replace one OS with another without deleting all the installed operating systems.

PINN workflow is very similar to NOOBS, first it needs to be installed on the SD card, once that is done the installation and maintenance of other OSes can be done on the RPI, without having to remove the SD card.

On first boot, PINN formats the SD card and enters in the recovery application, which allows to select the desired OSes from a list. The OS list is retrieved from locally available OSes (OSes need to be copied to the SD card before booting to use this feature) and those available from a remote repository.

On any subsequent boot, the recovery application can be accessed again by pressing SHIFT, Left mouse button, any key on a CEC enabled TV remote, or touch the Raspberry logo on the touchscreen.

If no input is provided, the boot selection dialog shows up with a list of the already installed OSes. If only one OS is installed, it will be booted immediately.

2.4 Berryboot

Figure 2.2: Berryboot Graphical Interface.

Berryboot is a multi-boot manager created by Floris Bos in C++ that allows to select and install multiple operating systems on a unique SD card to use with the Raspberry Pi [13].

The big difference between Berryboot and the other operating system selectors is that it shares the kernel between all the installed operating systems [14] making the list of possible operating systems much shorter when compared to NOOBS for example.

Berryboot uses a buildroot image which contains only the necessary to use the QT graphical interface where the user can select and install the operating systems.

Just like PINN it offers the possibility to install operating systems to external storage but has no possibility to install without access to screen, keyboard and mouse at least for the initial setup.

2.5 Rufus

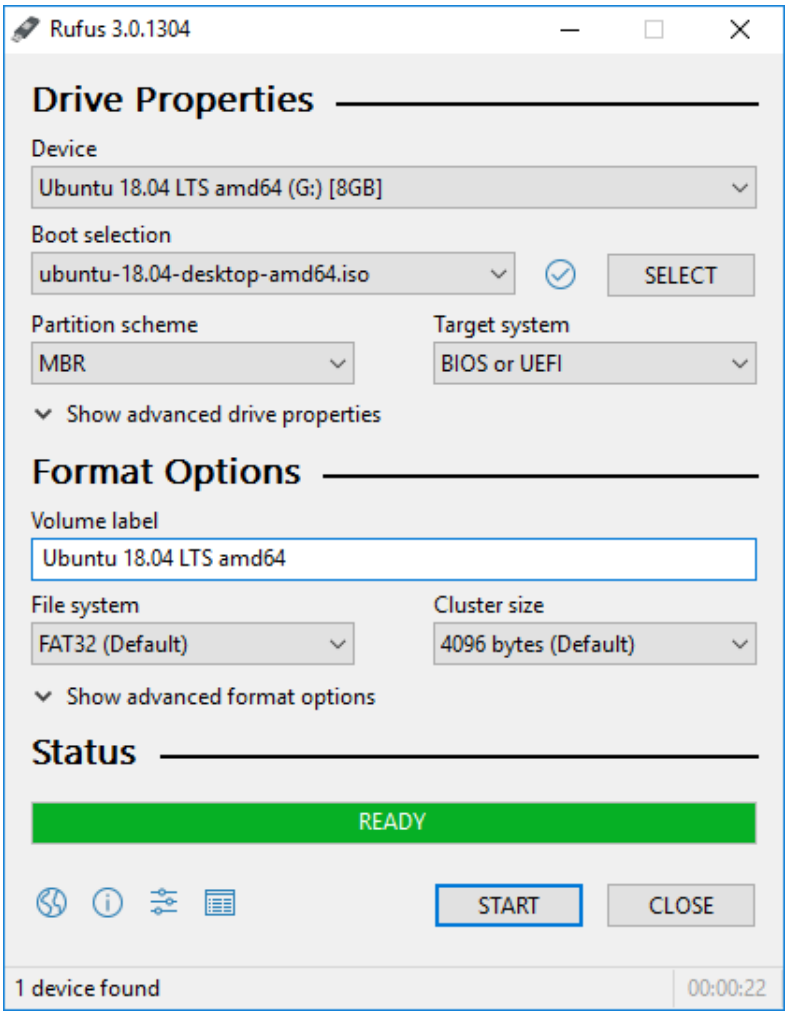


Figure 2.3: Rufus Graphical Interface.

Rufus is a utility created to help format and create bootable USB drives to install operating systems on other devices [15]. Was originally designed with only MS-DOS support on December 11, 2011. with the version 1.1.0 ISO image support was introduced and UEFI in version 1.3.2. If needed, it can install a bootloader such as SYSLINUX or GRUB onto the flash drive to make it bootable. It can be especially useful in the following cases:

- Create USB installation media from bootable ISOs (Windows, Linux, UEFI, etc.)
- Work on a system that doesn't have an OS installed
- Flash a BIOS or other firmware from DOS(Disk Operating System)
- Run a low-level utility

It's a very used tool specially due to its small size, fast operation and free license. Despite being largely distributed it's a open source project where some bugs can be fixed by the community. It's main features are:

- Format USB, flash card and virtual drives to FAT/FAT32/NTFS/UDF/exFAT/ReFS/ext2/ext3
- Create DOS bootable USB drives, using FreeDOS or MS-DOS (Windows 8.1 or earlier)
- Create BIOS or UEFI bootable drives, including UEFI bootable NTFS
- Create bootable drives from bootable ISOs (Windows, Linux, etc.)
- Create bootable drives from bootable disk images, including compressed ones
- Create Windows To Go drives
- Create persistent Linux partitions
- Download official Microsoft Windows 8 or Windows 10 retail ISOs
- Compute MD5, SHA-1 and SHA-256 checksums of the selected image
- Perform bad blocks checks, including detection of "fake" flash drives
- Small footprint. No installation required.
- Portable
- 100% Free Software

Rufus is specially designed for desktop use and creation of bootable media, but it has some similarities to this dissertation by the way it uses images to create bootable devices. It only supports .iso files and formats all the selected device, making it impossible to install multiple operating systems simultaneously but can still be used as storage device with the image still installed.

2.6 DD

DD is a command-line utility for Unix operating systems with the primary purpose of converting and copying files [16]. By default, dd reads from stdin and writes to stdout, but these can be changed by using the if(input file) and of(output file) extending the uses of this tool. DD can be used for disk wipe, by writing zeros to it, make exact copies of memory blocks, drive benchmarks or even backup entire disks or partitions. When doing a backup or restoring DD can convert the blocks into a single .img file and encrypt it making it a lot easier to organize several images and making the more secure.

When using DD, several parameters can be changed, like the size of the read and write block, convert the file to a specific format, input flags, output flags or skip blocks.

The usage of DD has to be made cautiously because improper usage or entering a wrong value can wipe, destroy, or overwrite the data on a device.

2.7 PXE - Preboot Execution Environment

PXE describes a standardized client-server that boots a software assembly [17]. For PXE to work the client needs to have a NIC with PXE enabled and a network connection to retrieve the image.

Despite not being yet very used on Raspberry Pies, PXE was introduced on the RPI 3 B+ and is a very frequent choice for desktop operating system booting, installation and deployment.

The big advantages of PXE booting is that the client computer no longer needs any storage media since everything is retrieved over network. This also simplifies the process of installing and configuring OSes on multiple computers.

PXE depends on three protocols, being them DHCP(Dynamic Host Configuration Protocol),TFTP and NFS. DHCP is used to provide an IP address to the client. The client sends a DHCP broadcast stating that it needs to PXE boot which is caught by the server and replied with a suggested IP address to use and the information on how to PXE boot. The client then, starts communicating with the server using the received IP address. After that, TFTP is used to retrieve the bootstrap program, boot files, necessary drivers and finally the image.

When the bootstrap program is retrieved, it is transferred into the RAM. This is sufficient to get running a minimalistic OS like WindowsPE or a basic Linux kernel. This OS loads its own network drivers and boots the downloaded image.

After the device boot all the posterior necessary files are also downloaded over the network from the TFTP server.

NFS allows the user to access files over the network, in this case the server, in the same way the file was on local storage. The server also has to have NFS implemented so that the client can request access to data.

2.8 Buildroot

Buildroot is a tool that allows to build a complete and bootable Linux image for embedded systems using cross-compilation [18]. It uses a set of Makefiles and patches that simplifies the process of image creation and personalization, while giving compatibility to various computer architectures and instruction sets like x86, ARM, MIPS and PowerPC [19]. With the option of a GUI interface, the user can select each packages to enable or disable and build the required cross-compilation toolchain, create the root file system, compile Linux kernel images, generate bootloaders for the target system or any independent combination of these steps.

This tool offers several features to simplify its usage, like default configurations for several embedded boards such as Cubieboard, Raspberry Pi and SheevaPlug, multiple standard libraries as part of the toolchain and several compression mechanisms for the final image such as cramfs, JFFS2, romfs, squashfs and UBIFS.

2.9 Linux Watchdog

In computing, a watchdog is something that monitors a system for normal behaviour, performing a reset in case of failure to stablish the normal operation [20]. Usually a watchdog timer is used to detect and recover from malfunctions. This timer is regularly reset during normal operation preventing it from time-out, if this timer is not reset a signal is emitted triggering the action that was configured to, usually a reboot [21].

This timers are usually found in computers and embedded systems, being called WDT(Watchdog Timer), and is used to react to faults that humans could not in time or systems that cannot be easily accessed.

Just like a conventional watchdog, the Linux Watchdog monitors Unix-based systems automatically, rebooting it if the system hangs due to unrecoverable errors. The Raspberry Pi has it own hardware WDT embedded in the SoC(System on Chip), making this tool possible to use on this device.

The Linux Watchdog acts like a daemon which operates in the background, communicating with the Raspberry WDT at least once per minute and preventing the time-out. Beside error monitoring, this tool can also be used to check the status of hardware or software, like current CPU(Central Processing Unit) temperature, overload or even if a program is doing a specific timed action.

2.10 Theoretical Concepts

2.10.1 Communication Protocols

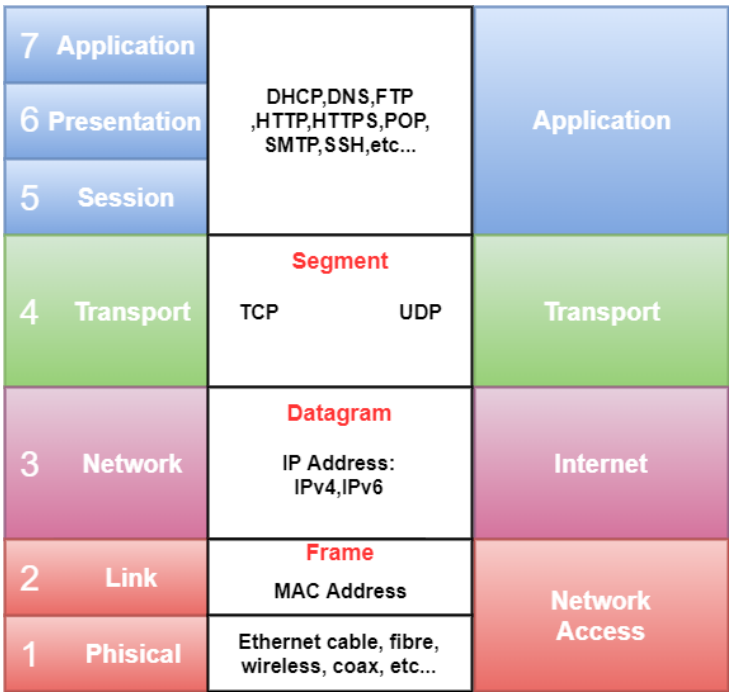


Figure 2.4: Communication protocols layer -

As already mentioned, one of the requirements for this dissertation is the communication between the Raspberry Pi and the central application, making the communication protocols an important part of this research. So, a trade off between the advantages and disadvantages of each protocol has to be made to know which one is the best and fits better on this dissertation.

One of the requirements established was that the communication had to be made over the internet, this restricts the list of possible protocols by a large quota. Since this protocol will be used on both the

desktop and Raspberry Pi, it has to be compatible with both devices and available operating systems, restricting even more the list of available protocols. Two different protocols have to be used, one for the communication between the central application and the Raspberry Pi and another for file transmission between the server where the images will be allocated and the Raspberry Pi.

2.10.1.1 TCP - Transmission Control Protocol

TCP is one of the main protocols of the internet originated in the initial network implementation to complement the Internet Protocol (IP) [22]. It provides reliable, ordered and error-checked delivery of a stream of bytes between applications running via an IP network. This protocol comes as an improvement to UDP which has much less reliability and is used by major internet applications like the World Wide Web, email, remote application and file transfer.

At lower levels of the protocol stack, several problems may occur like network congestion, traffic balancing, lost, duplicated or out of order packets. TCP can detect this problems and request the re-transmission of data or if the problem persists notify the source of the failure. With all of this levels of protection. TCP is known as accurate protocol but can incur relatively long delays while waiting for the correction of this problems. Therefore it is not the best option for real time applications like voice over IP.

To ensure that all bytes are received identically and in the same order as those sent, TCP uses a technique known as positive acknowledgement with re-transmission. This requires the receiver to respond with an acknowledge when receives data. The sender keeps record of each packet sent and has a timer to re-transmit a package when it expires, this avoids packet loss or corruption.

On TCP/IP protocol, IP handles the data delivery whilst TCP only keeps track of segments by dividing the a file into segments and forwarding them into the internet layer. The internet layer then encapsulates each TCP segment into IP segments by adding a header which contains information of the destination. When the packets are received the by the destination, the TCP software in the transport layer re-assembles the segments and assures they are correctly ordered and error-free and acknowledges the receipt.

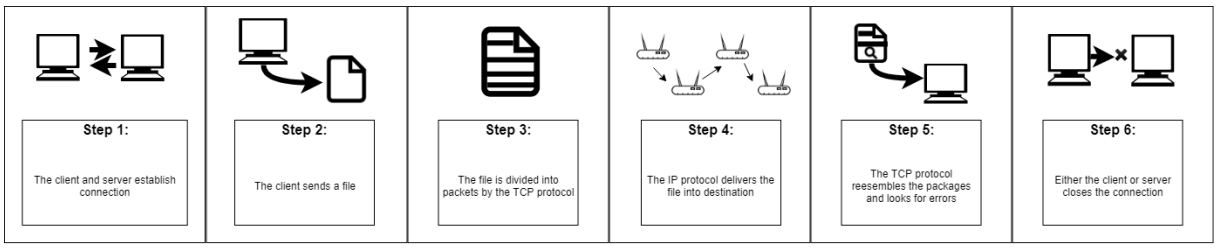


Figure 2.5: TCP/IP workflow.

The first phase of a TCP protocol is the connection, which must be established in a handshake process before starting transferring data. To establish a connection, TCP uses a three-way handshake, first the server needs to bind and listen to a port to open it up for connections. Once the port is open, the client can start the connection by sending a SYN to the server which is responded by a SYN-ACK from the server. The final step of the connection phase is the ACK(Acknowledge) from the client to the server which acknowledges the connection establishing a full-duplex communication.

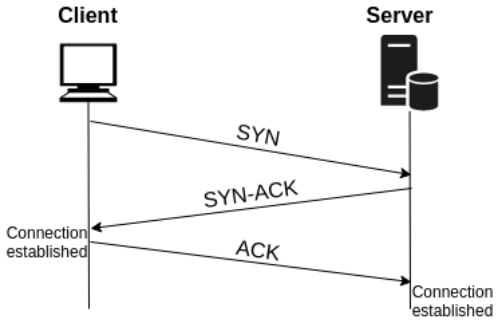


Figure 2.6: TCP/IP connection establishment.

The second phase is the data transmission, which is mostly done by the IP protocol. Here the TCP role is to divide the packages and add headers to which one of them so the TCP layer of the destination can rearrange the packages and check for errors.

The last phase of this protocol is the connection termination which can be done by any of the parts using a four/way handshake. When an endpoint wishes to stop the connection, transmits a FIN package which is acknowledged by the other end with an ACK. Usually, when the ACK for a FIN is sent it will go with a FIN package too so the end that wants to close the connection can also acknowledge with an ACK, ending the communication after a timeout and opening the port to other connections.

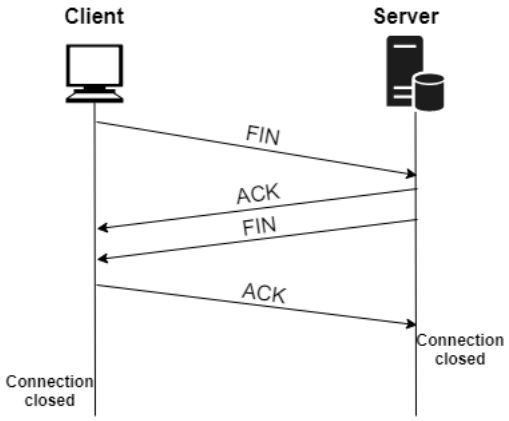


Figure 2.7: TCP/IP connection termination.

2.10.1.2 UDP - User Datagram Protocol

UDP is a protocol created by David P. Reed in 1980 that allows computer applications to send messages, called datagrams, to other host on an IP protocol network [23].

This protocol is usually used in time-sensitive applications where error checking and correction is not necessary, because losing packages is preferable than delay due to retransmission.

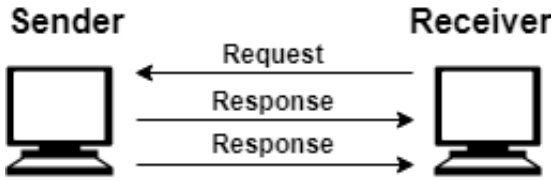


Figure 2.8: UDP workflow.

Thanks to a connectionless communication model with a minimum of protocol mechanisms, prior communications are not required, in order to set up communication between endpoints.This makes UDP a much simpler protocol than FTP where the only phases of it are the request followed by the sender responses.

The main features of this protocol are:

- Transaction-oriented, suitable for simple query-response protocols such as Domain Name Service.
- Provides datagrams, suitable for modelling other protocols such as IP tunnelling or remote procedure call.
- Simple, without a full protocol stack, such as the DHCP.
- Stateless, making it suitable for very large number of clients, such as media streaming applications.
- Lack of retransmission delays which makes it suitable for real-time applications such as Voice over IP and many protocols using Real Time Streaming.
- Supports multicast, making it suitable for broadcast information such as in many kinds of service discovery.

2.10.1.3 SCTP - Stream Control Transmission Protocol

SCTP is a communication protocol which operates at the transport layer and serves a role very similar to TCP and UDP providing features of both of this protocols [24]. It is message oriented like UDP ensuring

reliable, in-sequence transport of messages like TCP. It differs from those protocols by providing multi-homing and redundant paths to increase resilience and reliability. To prevent SYN flooding attacks, and improve security, SCTP implemented a four-way handshake connection method.

The connection starts with the client sending a INIT packet. The server responds with a with a INIT-ACK which includes a COOKIE (a unique identifier generated by the server). The client then responds with a COOKIE-ECHO which contains the identifier previously sent. At this point, the server starts allocating the resources for the connection and sends a COOKIE-ACK, connecting both endpoints.

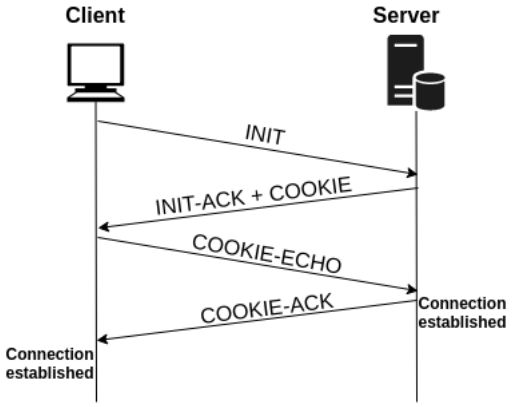


Figure 2.9: SCTP connection establishment.

In TCP, both endpoints can close the connection, but until the socket is closed it can continue to transmit data, resulting in a half-closed state. To avoid this, STCP uses a different termination method. When a SHUTDOWN packet is sent, both endpoints are required to close the connection, avoiding further data movement.

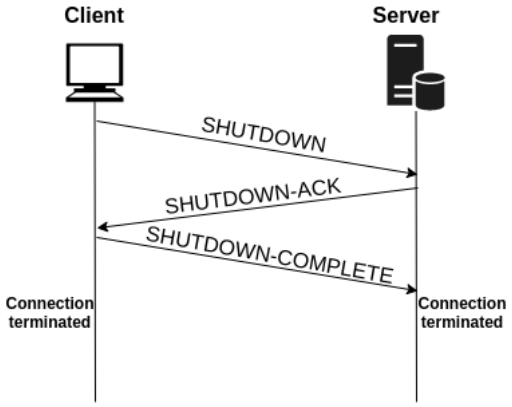


Figure 2.10: SCTP connection termination.

The main features of SCTP include:

- Reliable transmission of both ordered and unordered data streams.

- Multi-homing support in which one or both endpoints of a connection can consist of more than one IP address, providing redundant network paths.
- Delivery of chunks within independent streams eliminates unnecessary head-of-line blocking.
- Explicit partial reliability.
- Path selection and monitoring providing a way to test the connectivity of the transmission path.
- Validation and acknowledgement mechanisms to protect against flooding attacks and provide notifications of duplicated or missing data chunks.
- Improved error detection

The biggest difference between this protocol and TCP is that it can divide the streams of bytes into chunks contrarily to the unbroken stream as does TCP. As in UDP, SCTP transfers messages in one operation with the reliability and order of TCP.

When applications submit their data to be transmitted in messages, SCTP places this messages and control information into separate chunks, each identified by a header. The protocol can fragment messages into a number of data chunks, but each one of them contains data only from one user message. This chunks are then bundled into packets containing a packet header, SCTP control chunks and data.

2.10.1.4 FTP - File Transfer Protocol

FTP is a network protocol used for transferring files built in a client/server model architecture via network [25]. This protocol, uses separate control and data connections between the client and server which allows the users to authenticate with a clear-text sign-in protocol. This sign-in is usually done in form of a username and password, but also allows anonymous connections if the server is configured for it.

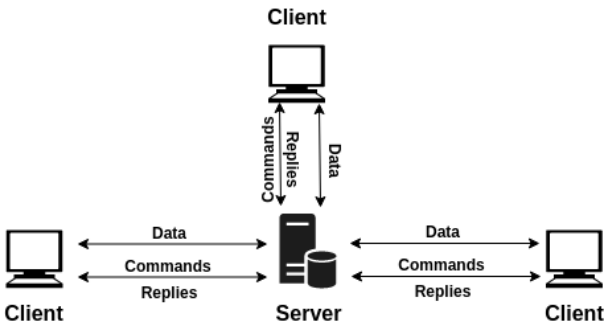


Figure 2.11: FTP workflow.

Due to the lack of security when transferring the username, password and data, FTPS and SFTP were created, making FTP obsolete.

FTPS added support to TLS(Transport Layer Security) and SSL(Secure Sockets Layer) adding encryption to the FTP connection [26]. It offers two separate methods to invoke client security, implicit and explicit. While implicit method requires that the security is established from the beginning of the connection in the explicit method, the client has to request the security from the server and agreed what will be the encryption method used, giving full control over what areas of the connection that will be encrypted. Like HTTPS, FTPS servers must provide public key certificate signed by a trusted certificated authority. This assured that the client is connected to the requested server avoiding man-in-the-middle attacks.

SFTP uses the SSH(Secure Shell) protocol to transmit and encrypt data [27]. This gives several capabilities to this protocol such as resuming interrupted transfers, directory listing, and remote file removal. Some dedicated FTP servers implement this protocol, however, usually am SSH server implementation is required, as it shares the SSH port with other SSH services.

2.10.2 Raspberry Pi boot sequence

The Raspberry Pi boot sequence is different from a desktop, having its own sequence divided in three stages.

Initially the CPU is held in reset, giving the responsibility of booting the system to the Videocore GPU. It loads the first stage bootloader from a ROM(Read-only Memory) embedded in the SoC. This stage consists in finding the first partition in the SD Card, finding the bootcode.bin, load it to the L2 cache and run it [28]. After this, the second stage starts. This first partition has te be FAT for the GPU to properly read it.

When bootcode.bin is loaded, the Videocore GPU runs it and looks for start.elf, starting the third stage bootloader [28]. Most of the boot processes are made in this stage, starting with config.txt, a text file containing configuration parameters for the GPU and Linux Kernel [28]. Once this file is loaded and parsed, cmdline.txt and kernel.img are next files being loaded [28].

Only on this stage the CPU is released from reset, first the cmdline.txt is read which contains the kernel command parameters needed for the boot and then kernel.img is loaded into the shared memory allocated to he ARM processor [28]. The start.elf file can also contain some extra parameters for the kernel. After all the parameters are loaded, the kernel starts booting finishing the boot sequence.

In image 2.12 is described the workflow of the Raspberry Pi boot.

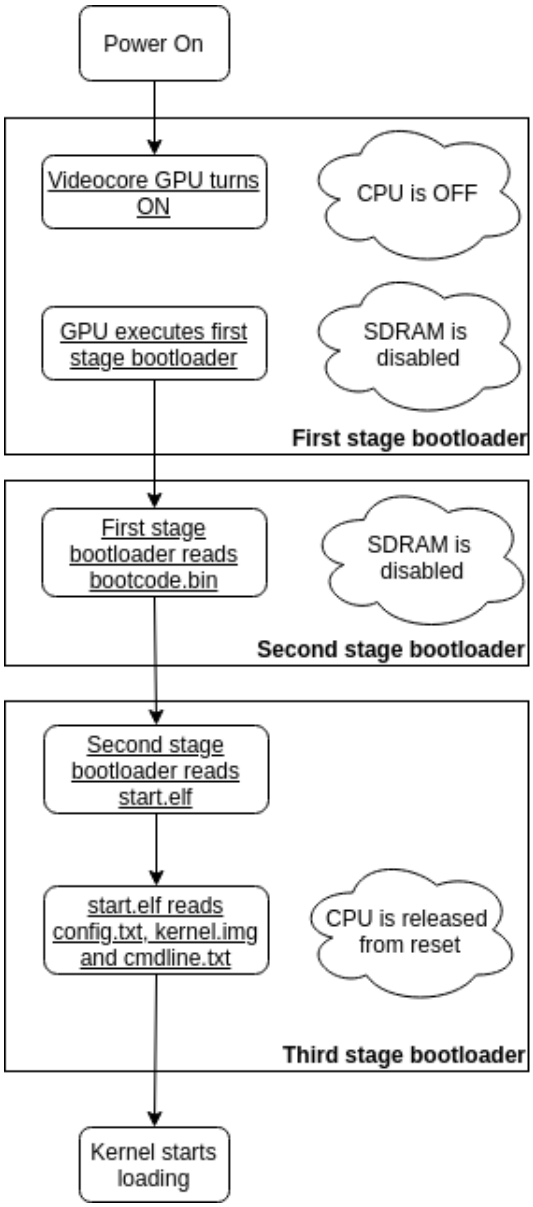


Figure 2.12: Raspberry Pi boot sequence.

2.10.3 Disk Partitioning

Usually every drive can be sliced into several regions, giving the possibility to manage each one separately. Each region is called partition and is stored in a partition table, containing the information and order of each of the partitions. Partitioning allows to install multiple operating systems on a single drive, being each one of them Independent between each other.

With DOS and Windows, usually one primary partition is used for the file system, which contains the operating system,swap file, utilities, applications and user data. This partitions contain also a drive letter to identify it.

On Unix-based operating systems usually two partitions are needed, one containing the Linux kernel called boot and another with the system itself, called root.

2.10.4 Daemon

A Daemon is a computer program that runs as a background process without the direct control or interaction with the user [29].Normally daemons are started a when a system boots and unless forcibly terminated, run until shutdown providing services either for the system or user programs.

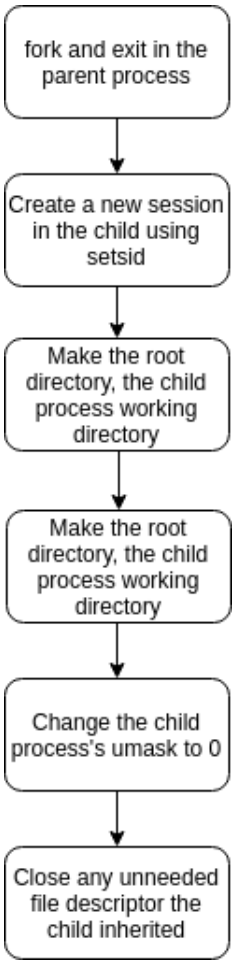


Figure 2.13: Daemon creation.

The creation of a daemon has very simple steps. Initially the parent process needs to fork and exit creating a child causing init to adopt this process. After this, the child process needs to create a new session using the setsid call and make his working directory the root directory. With this, the daemon gets superuser privilege which is needed due to the duties that it usually performs.

The last step is to set the daemon umask to zero, this will prevent interference with the creation of files and directories.

As already said, a daemon has no direct interaction with the user, so any unneeded file descriptors need to be closed like stdin, stdout and stderr.

2.11 Conclusion

This chapter gives a general overview of tools and utilities similar to this dissertation giving also an idea of how groundbreaking this dissertation can be. With this research it was also possible to understand some topics needed for the development of the OS installation, the available communication protocols and how the partition structure can be.

After doing the state of art research it is possible to verify the existence of some Raspberry Pi Operating system installers which do some of the desired requirements but none of them fulfils all of them. Since bootcode.bin and other files necessary for the Raspberry to boot are still close-source and there is not many information about it, the main approach for bootloader creation is to use a minimal operating system with all the necessary to run a GUI and do what is expected. This operating system can be done with Buildroot or Yocto due to the feature that allows to select the desired packages and options, minimizing it to the maximum.

PINN is the closest one to what is wanted, giving the possibility to select and install OSes locally or remotely and having a heedlessly way of being used via VNC (Virtual Networking Computing). While some of the main requirements are met, the option to connect to another repository, the possibility of managing both the installed OS and the recovery GUI via commands and the fault tolerance system are not implemented. Although it gives a big advance to what is developed on this dissertation.

It is also possible to verify the large variety of desktop OS installers, but they don't give the option to change the OS without reformat the SD card and inserting it again on the desktop.

Another option found which allows to remotely install operating systems without the necessity of using SD cards is the PXE, but is still a very early option on the Raspberry Pi and is still being developed better options to use it, but is still a great option to be developed in future work.

The chosen option to start developing this dissertation was NOOBS, so part of what was done in this dissertation was inspired on it. Being NOOBS an open-source project and the base to PINN development, it is easy to check the source code and use the buildroot configuration to generate the minimal image needed for the manager. The installation process, the partition table idea and the buildroot image was developed using the NOOBS source-code.

For the file transfer protocol the one that was chosen was the FTPS, specially due to the compatibility of the wget command with it. As already mention above this protocol offers some security with SSL/TLS making the connection much more secure. It was chosen rather than SFTP specially due to being a much

simpler protocol and not requiring SSH to be enabled.

One of the constraints of this project is to prevent fault tolerance, with this, the system needs to be monitored to prevent hang outs. During the research of tools that could be used to prevent this, Linux Watchdog was found, complementing all the needs in this point. With this, was decided to not implement a new watchdog, using the open-source option from Debian.

With this, we assure that what was developed in this dissertation is currently groundbreaking specially by the remote option which significantly decreases the necessity of interacting directly with the Raspberry Pi, making this application very useful for deployed devices.

Chapter 3

System Architecture

3.1 Introduction

In this chapter is possible to read about the System Specifications and all the chosen modules designed to build the system. The chapter also explains some choices about the modules, what is composed by, and the requirements and constraints taken into account.

In the beginning of this chapter is provided an general overview of this dissertation. Giving the reader an idea of how this system fits, and giving an example of use case.

Despite security being a very important of the general system, it was not taken into consideration the design of specific security protocols or methods, as it will be done outside of this dissertation.

It will be presented three subsections about the Operating System Installer, the Daemon and the Configuration Scripts with details on what programs and packages were used, their versions and why.

In the end of this chapter a brief look of the protocols used will also be given.

With all the information provided in this chapter the reader should be able to develop everything that was done in this dissertation.

3.2 General Overview

This system is part of a wider architecture composed by several systems. What was developed on this dissertation is meant to be specially used in a group of several deployed Raspberry Pi's, also giving the opportunity for those who want to use it more personally and a single device.

In addition to he Raspberry Pi's, is necessary a central application that can manage and control all the connected devices via internet. This application should be able to connect to the Raspberry Pi's, send commands to them, check their status, connect to FTP servers and check and manage the files on them.

To save all the necessary images, a FTP server is also necessary. This server should be able to be connected by the Raspberry Pi's and send the requested files.

This dissertation will only take into account the software developed for the Raspberry Pi's, including the Remote Boot Manager, the daemon and all configuration files necessary for the Raspberry.

In the image 3.1 is a picture of the general architecture where this system can be implemented.

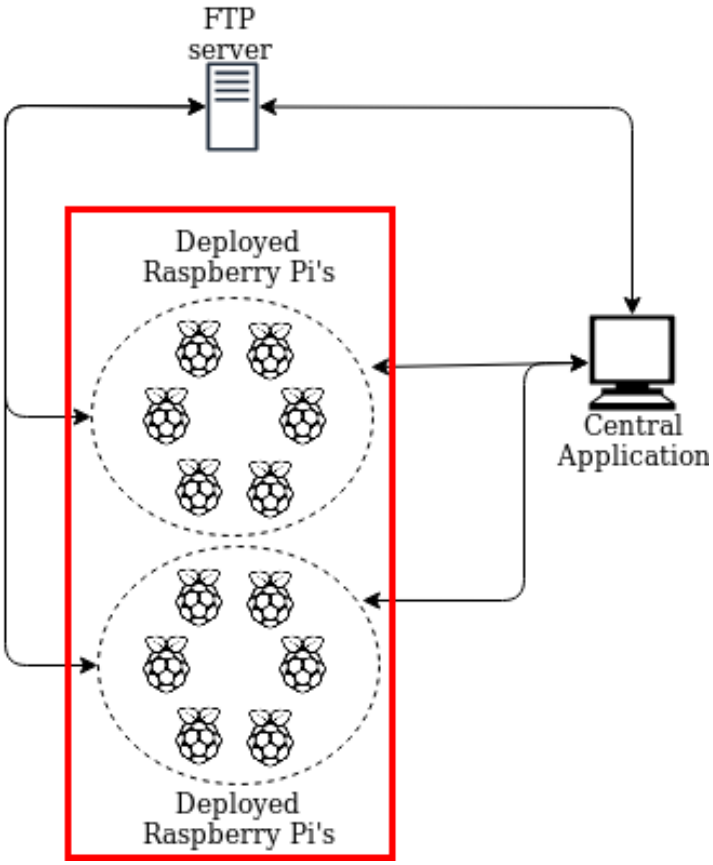


Figure 3.1: General Overview schematic.

Is important to remember that the Raspberry Pi's have no need to communicate between them, because all the commands should come from the Central Application.

3.3 System Requirements

- **Be completely configurable headlessly:** The system has to have a way to configure all parameters before the system boot so it can connect to the central app automatically and receive commands from it without any kind of peripherals plugged in.
- **Compatibility with multiple Operating systems:** The system should be compatible with multiple Operating Systems to install them.
- **Fault tolerance:** The system should have ways to recover from errors at kernel or operating system level, either by rebooting or surpassing those errors.
- **Controllable by commands:** The system should be controlled by commands received from the Central App.

3.4 System Modules

The system is divided in two different modules, being them separated by the before operating system installation and the after.

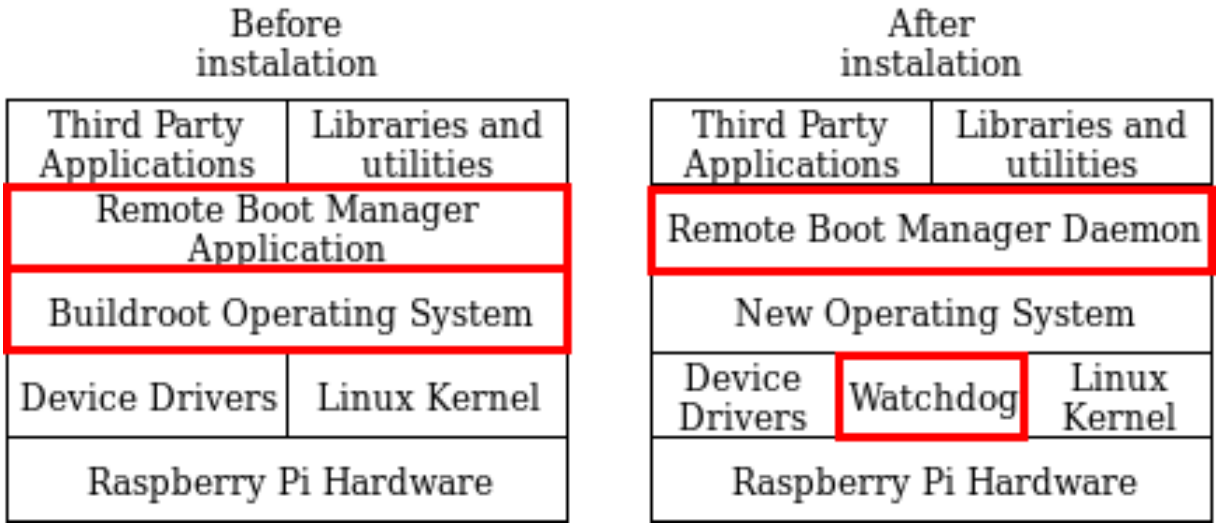


Figure 3.2: System modules schematic.

Before an operating system being installed the system will start in the Remote Boot Manager application which will receive commands from the central application until an operating system be installed. This application uses an operating system created used Buildroot and a custom kernel

After it is installed the freshly installed operating system will be started with a daemon and a watchdog inserted in it. This daemon will allow the Central App to communicate with the device and check its status. The watchdog will be responsible to check for system errors and reboot it if necessary.

The new operating system can have any Linux based operating system and kernel.

3.5 System Specification

The device selected for this master's dissertation was the Raspberry Pi 3B+ and the Raspberry Compute Model 3. This device was selected because it was the most recent Raspberry Pi by the moment of this dissertation and one of the most used IoT devices.

In the general project specification, several Raspberry Pi devices are used, allowing the Central Application to control each one of them independently.

In table 3.1 is the specification of the Raspberry Pi 3B+.

Table 3.1: Raspberry Pi 3B+ specification

Model:	Raspberry Pi 3B+
SoC:	Broadcom BCM2837B0 quad-core A53 (ARMv8) 64-bit @ 1.4GHz
GPU:	Broadcom Videocore-IV
RAM:	1GB LPDDR2 .
Bluetooth:	Bluetooth Low Energy 4.2 .
Ethernet:	Gigabit Ethernet 2.4GHz
Wi-Fi:	5GHz 802.11b/g/n/ac.
Storage:	SDCard.
GPIO:	40-pin GPIO header.
Ports:	HDMI, 3.5mm analogue audio-video jack, 4x USB 2.0, Ethernet, Camera Serial Interface (CSI), Display Serial Interface (DSI).
Dimensions:	82mm x 56mm x 19.5mm, 50g.

3.5.1 Python

In this dissertation, Python was specially used to develop the daemon that communicates with the Central App after an OS ins booted. The version selected was 2, due to the fact that the daemon will be copied into every newly installed OS and this version is more commonly encountered already installed, avoiding incompatibility problems. This version is also described as the more stable one and has better support than the Python 3 versions. After the selection of the version, the necessary packages were also selected. This packages were chosen taking into account what was necessary for the Daemon development and what would speed up the Daemon start. In table 3.2 is shown the respective description of each used package :

Table 3.2: Python packages

Package	Description
socket	Provides access to the BSD socket interface
errno	Provides access to the error number of system symbols
string	Provides multiple access to functions to handle strings
daemon	Implements the well-behaved daemon specification of PEP 3143
os	Provides a portable way of using operating system dependent functionality
fcntl	Performs file control and I/O control on file descriptors
platform	Gives access to several platform informations
subprocess	Allows to spawn and control new processes
time	Provides various time-related functions
getnode	Gives access to the MAC(Media Access Control) address number of the system

3.5.2 Pyinstaller

Pyinstaller is a package that reads python scripts, analysing the code and discovering every module and library needed to execute it [30]. After that, it collects a copy of each of those files, including the python interpreter, and puts everything in a single folder or in an executable file.

This can be later distributed and executed without the need of installation of any python version or modules.

This program will be used in this dissertation to generate the daemon as an executable file, making it easier to inject on the newly installed operating systems by the Remote Boot Manager application. With

this method, the user is relieved from the work of downloading, installing and configuring all the necessary packages for the daemon, whilst making it portable and multi-platform.

3.6 Partitioning

Partitioning is a very important specification in this dissertation. Knowing the boot sequence from the Raspberry Pi was possible to define how the partition table needed to be.

The first partition should always be the Operating System Installer. This allows the creation of a bootloader because the first partition is always the one that is booted.

According to the Raspberry Pi boot sequence, the first partition should also always be a FAT partitioning.

To simplify the way the user can start using the Operating System Installer, the only partition necessary on first boot is a FAT one containing only the Remote Boot Manager files.

Table 3.3: Partitioning on fresh install

Partition Number	Type	Label	Contents
1	FAT	New Volume	Remote Boot Manager freshly installed files

After the first boot, when the Raspberry Pi first enters the RBM(Remote Boot Manager) graphical interface, some steps are done, changing the partition table. The first partition is shrunken to fit only enough to contain the RBM files and is named 'RECOVERY'. By never writing to this partition is possible to limit this size , needing anything else.

Another partition, that will contain all the installed operating systems, has to be created. This partition has to be an extended partition to work around the limitation of 4 primary partitions.

The last partition added is where the configuration files are stored, this allows to change any parameter contained in the files by simply plugging the SD Card on a computer or accessing this partition, making this process much simpler. The structure of the partition table is described in table 3.4.

Table 3.4: Partitioning after first boot

Partition Number	Type	Label	Contents
1	FAT	Remote Boot Manager files	Remote Boot Manager freshly installed files
2	extended		Logical partitions of new Operating Systems
3	ext4	SETTINGS	RBM settings files

When an operating system is about to be installed, both the root and boot partitions are created as logical partitions of the extended one. This allows in the future to implement dual boot without having to change anything. The final structure of the partition table is described in the table 3.5.

Table 3.5: Final partitioning

Partition Number	Type	Label	Contents
1	FAT	Remote Boot Manager files	Remote Boot Manager freshly installed files
2	extended		Logical partitions of new Operating Systems
5	FAT	boot	New operating system boot files
6	ext4	root	New operating system root files
3	ext4	SETTINGS	RBM settings files

3.7 Communication Language

To restrict the number of possible commands between the Central Application and the Raspberry Pi, a language has been designed. This language specifies the structure that the commands should have, so that any endpoint can easily parse and execute them.

There are five main commands that can be sent, which are identified by a starting # symbol. These commands are #command, #commandoutput, #status, #newimage, #getinfo, #info.

Some of these commands can have sub-commands, which are identified by a $ symbol. Each sub-command is followed by a variable parameter which the user can change.

The table 3.6 specifies all the possible parameters of this commands and some examples.

Table 3.6: Command list table

Command:	Sub-Command:	Example:	Direction:
#command		#command cd /	Central Application-RPI
#commandoutput		#commandoutput Shutting down in 10 sec	RPI-Central Application
#status		#status Rebooting	RPI-Central Application
#getstatus			Central Application-RPI
#newimage	$IP $USER $PW $PATH	#newimage $IP 10.42.0.1 $USER ftpuser $PW ftp $PATH /Raspbian	Central Application-RPI
#getinfo			Central Application-RPI
#info	$MAC $OS $ID	#info $MAC ASD123123-asdasdad $OS Raspbian $ID RPICMDL	RPI-Central Application

The #command command, allows the user to execute any terminal command in the Raspberry Pi, like for example change directory or reboot. As a response to this command, the RPI can send #command-output if the executed command has output.

While the application continues its execution, periodically or when the state changes, the RPI will send a #status command, informing the Central Application of the status of execution, like "Rebooting". This command is triggered by receiving #getstatus from the Central Application.

If the user wants to install a new operating system, a #newimage command needs to be sent to the RPI. This command has several sub-commands that also need to be sent, being them the IP, username and password of the FTP server and the path where the new image is located in the server.

When the RPI connects to the Central Application, a #info command is sent, with this the RPI identifies itself. As demonstrated above, this command has three sub-commands, which are the RPI MAC address, the current operating system and the ID that identifies the RPI. The Central Application can request this information at any time by sending a #getinfo command.

By designing a communication language, the security is improved, because any endpoint needs to know it in order to communicate, filtering any message that does not follow this structure.

3.8 Remote Boot Manager Application

When the system boots, the Remote Boot Manager Application should start, due to being saved in the first partition. It acts like a bootloader by starting as earlier as the system boots, with the difference that it lays on top of an operating system.

The framework selected to develop the graphical user interface was QT, due to its ease of use and large wide of libraries.

For the creation of the operating system, the framework selected was Buildroot. It allows to create an OS with only the necessary packages and compressed with several formats, reducing the space to the maximum.

Below is the requirements this application should meet:

- **Connect with the Central Application:** The application should be able to connect to the Central Application to receive commands from it.
- **Be controlled via commands:** The application should be able to receive and be controlled entirely by commands received from the Central Application.
- **Be controlled without connecting to the Central Application:** The application should have ways to be used without connecting to the Central Application.
- **Connect and receive files from FTP servers:** The application should be able to connect to different FTP servers in order to receive the new operating system.
- **Be able to install different operating systems:** The application should be able to install and boot different operating systems.
- **Communicate the status to the Central Application:** The application should be able to communicate to the Central Application its status, either periodically or when receiving a specific command .
- **Assure the system can still receive commands after the installation of a new operating system:** The system should be controlled by commands received from the Central App.
- **Detect the installation of operating systems and start them automatically:** The application should be able to detect if an operating system is already installed when booting and start it automatically.

One of requirements the application should meet is the possibility of using it headlessly (without monitor and keyboard), for this configurable configuration files have to be used. Also, it should provide a way for the freshly installed operating system communicate with the Central Application. This is done by injecting a daemon in the file system which will run on boot time.

3.8.1 Buildroot

To facilitate the development of the Remote Boot Manager application, the system is built on top of an operating system made using Buildroot.The chosen version for this dissertation was 2015-02.

Since we are building the operating system for Raspberry Pi, it necessary to change the configuration of Buildroot in order to build it for the correct device.

In target options, is possible to select parameters related to the device, which in this case is the Raspberry Pi. Here the user can select the architecture, binary format or instruction set that corresponds to the device the image is being build for.

In table 3.7 is possible to find the chosen target options configuration:

Table 3.7: Buildroot target options

Location	Configuration	Value	Description
Target Options	Target Architecture	ARM (little endian)	Target architecture family to build for
Target Options	Target Binary Format	ELF	Target binary format to build for
Target Options	Target Architecture Format	arm1176jzf-s	CPU variant to use
Target options	Target ABI	EABI	Application Binary to use
Target options	Floating point strategy	VFPv2	Strategy to use for floating point
Target options	ARM instruction set	ARM	Instruction set to be used

In the toolchain tab, the user can select options related to the image that is being built. Here is possible to select the toolchain type, compiler options, thread options, and languages to use in the system.

In table 3.8 is possible to find the chosen toolchain options configuration.

Table 3.8: Buildroot toolchain

Location	Configuration	Value	Description
Toolchain	Toolchain type	Buildroot toolchain	Which toolchain to be used
Toolchain	Kernel Headers	Linux 3.18.x kernel headers	Version of kernel header files to be used
Toolchain	C library	uClibc	C library to be used
Toolchain	uClibc C library version	uClibc 0.9.33.x	Version of uClibc to be used
Toolchain	Enable large file support	true	Enable option for toolchain to support files bigger than 2GB
Toolchain	Enable WCHAR support	true	Option to enable toolchain to support wide characters
Toolchain	Compile and install uClic utilities	true	Option to compile and install uClibc utilities to the target
Toolchain	Binutils Version	binutils 2.24	Version of binutils to be used
Toolchain	GCC compiler Version	gcc 4.8.x	Version of GCC to be used
Toolchain	Enable C++ support	true	Enable and install C++ libraries on target system
Toolchain	Enable compiler tls support	true	Option to enable the compiler to generate code for accessing thread local storage variables
Toolchain	Purge unwanted locales	true	Specify what locales to install on target
Toolchain	Enable MMU support	true	Specify if target has MMU

In the system configuration tab, is possible to change configurations related to the filesystem like password, username, init system, overlays and post scripts.

In table 3.9 is possible to find the chosen system configuration options.

Table 3.9: Buildroot system configuration

Location	Configuration	Value	Description
System configuration	Password encoding	md5	Password encoding scheme to encode passwords
System configuration	Init system	BusyBox	What init system to use
System configuration	/dev management	Dynamic using devtmpfs only	What type of /dev management to use
System configuration	Root F5 skeleton	default target skeleton	Type of skeleton to use
System configuration	/bin/sh	busybox default shell	Which shell will provide /bin/sh
System configuration	Run a getty after boot	true	Option for running getty after boot
System configuration	remount root filesystem read-write during boot	true	remount the filesystem as read-write during the boot process
System configuration	Root filesystem overlay directories	Location of the folder with the daemon and initialization scripts	List of directories that are copied over the target root filesystem after the build has finished

In the kernel tab, is possible to change options related to the kernel such as version, configuration and binary format.

In table 3.10 is possible to find the chosen kernel configuration.

Table 3.10: Buildroot kernel

Location	Configuration	Value	Description
Kernel	Linux Kernel	true	Option to build a Linux kernel for your embedded device
Kernel	Kernel version	Custom git repository	Location to get the kernel to be used
Kernel	URL of custom repository	git://github.com/ raspberrypi/linux.git	URL for custom kernel
Kernel	Custom repository version	12d78096b1669a08 d440f7ebaddf5d9 25e52fe79	Revision to use in the typical format used by Git
Kernel	Custom kernel patches	linux-regdb.patch	A space-generated list of patches to apply to the kernel
Kernel	Kernel configuration	Using a custom config file	Option to use costum config file
Kernel	Configuration file path	kernelconfig-recovery.armv6	Path to the kernel configuration file
Kernel	Kernel binary format	zImage	Format to be used by kernel
Kernel	Build a Device Tree Blob	true	Option to build DTB
Kernel	Device tree source	Use a device tree present in the kernel	Which device tree to use

With this tool, is possible to select the necessary packages, avoiding what is not necessary and saving space. In table 3.11 is a list of the necessary packages for this system and a description of what they are used for:

Table 3.11: Buildroot target packages

Location	Package	Description
Target packages	BusyBox	Software suite that provides several Unix utilities in a single executable file
Target packages	arora	Arora web browser
Target packages	p7zip	Installs 7z
Target packages	partclone	Only installs partclone.restore
Target packages	recovery	recovery GUI
Target packages	rpi-wifi-firmware	Provides the wifi firmware for the Raspberry Pi

Is also possible to select the filesystem configuration and required utilities. In table 3.12 is a list of the chosen configuration and a description of each option:

Table 3.12: Buildroot filesystem and flash utilities

Location	Package	Description
Target packages/Filesystem and flash utilities	dosfstools	Tools for creating and checking DOS FAT filesystems
Target packages/Filesystem and flash utilities	fsck.fat	Check a DOS filesystem
Target packages/Filesystem and flash utilities	mkfs.fat	Creates a DOS filesystem
Target packages/Filesystem and flash utilities	e2fsprogs.fat	Ext2 file system utilities
Target packages/Filesystem and flash utilities	e2fsck	
Target packages/Filesystem and flash utilities	e2label	
Target packages/Filesystem and flash utilities	fsck	Check a DOS filesystem
Target packages/Filesystem and flash utilities	mke2fs	
Target packages/Filesystem and flash utilities	resize2fs	
Target packages/Filesystem and flash utilities	mtools	Collection of utilities to access MS-DOS disks from Unix without mounting them
Target packages/Filesystem and flash utilities	ntfs-3g	read/write driver for Linux
Target packages/Filesystem and flash utilities	ntfsprogs	Install NTFS utilities

As specified, QT was the framework chosen to build the GUI. To compile and use QT, many dependencies need to be selected, so the majority of the graphic libraries selected are necessary for QT. In table 3.13 is a list of these libraries and a description of each of them.

Table 3.13: Buildroot QT dependencies

Location	Package	Description
Target packages/Graphic libraries and applications	Qt	Cross-platform application and UI framework
Target packages/Graphic libraries and applications	qjson	Qt library for mapping JSON(Javascript Object Notation) data to QVariant objects
Target packages/Hardware handling	rpi-firmware	Pre-compiled binaries of the current bootloader and GPU firmware
Target packages/Hardware handling	dbus	D-Bus message bus system
Target packages/Hardware handling	parted	GNU partition resizing program
Target packages/Hardware handling	rpi-userland	Library to use the Videocore driver
Target packages/Networking applications	dhcpcd	RFC2131 compliant
Target packages/Networking applications	wget	Network utility to retrieve files
Target packages/Networking applications	wpa-supplicant	WPA supplicant for secure wireless networks

To install and operate with the OS's tools like mount/ umount are used. This tools need to be selected on the buildroot configuration file to be included in the final image. A list of the chosen tools and a description of each them can be found in table 3.14.

Table 3.14: Buildroot Linux tools

Location	Package	Description
Target packages/System tools	util-linux	Various useful linux libraries and utilities
Target packages/System tools	mount/umount	mount/umount filesystems
Target packages/System tools	partition utilities	Partition tools (addpart,delpart,partx)
Filesystem images	squashfs root filesystem	Build a squashfs root filesystem

Using post-scripts is also possible to insert the compiled Remote Boot Manager application in the image and configure it to start on boot.

Is important to notice the usage of overlays, where the daemon and initialization scripts are. This allows to copy this files to the filesystem where the Remote Boot Manager application is, so that it can use them.

Is also possible to notice that a custom kernel is being used. This kernel is stored in a git page and was used to reduce the necessary space.

Two very important packages are QT and BusyBox. QT is the framework that is going to be used to build the application, whilst BusyBox is the system that allows to access the operating system used by the application.

With all the options indicated above, is possible to replicate the operating system used in this dissertation.

3.8.2 Graphical User Interface

When the application starts the first window is the Main Window, here the application checks if an Operating System is installed before showing. If there is an Operating System installed, the system will start a countdown showing the Operating System that will be booted, giving the chance to the user to cancel this operation.

If no Operating System is already installed, the Main Window will show. Here the user can input an IP and port to connect to the Central Application, the IP of an FTP server and check information about the disk. Image 3.3 shows a graphical representation of this window.

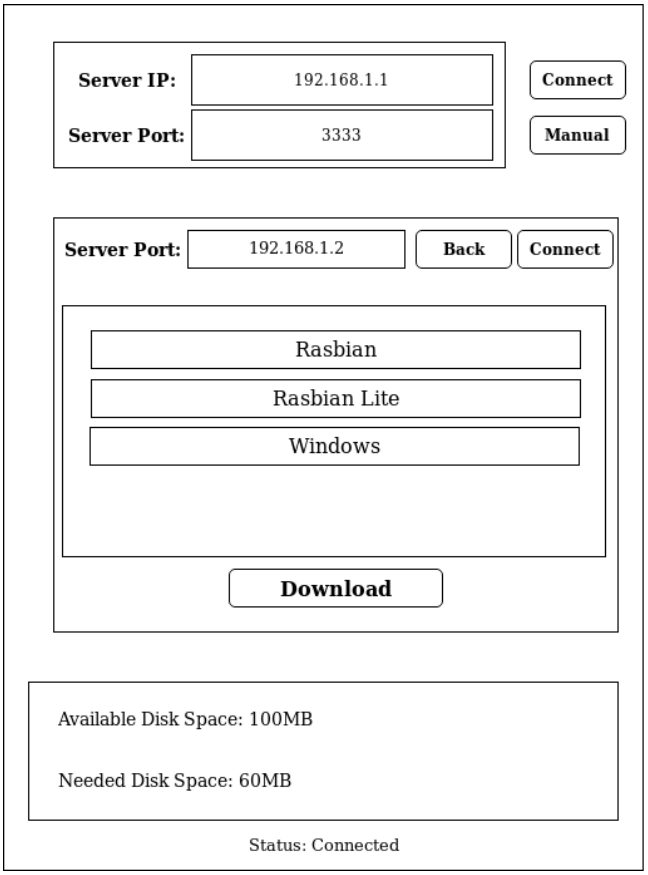


Figure 3.3: MainWindow design.

When the user selects an Operating System from the FTP tab or if a command to install a new one is received, a new window will show displaying the installation status. A graphical representation of this window can be found in image 3.4.

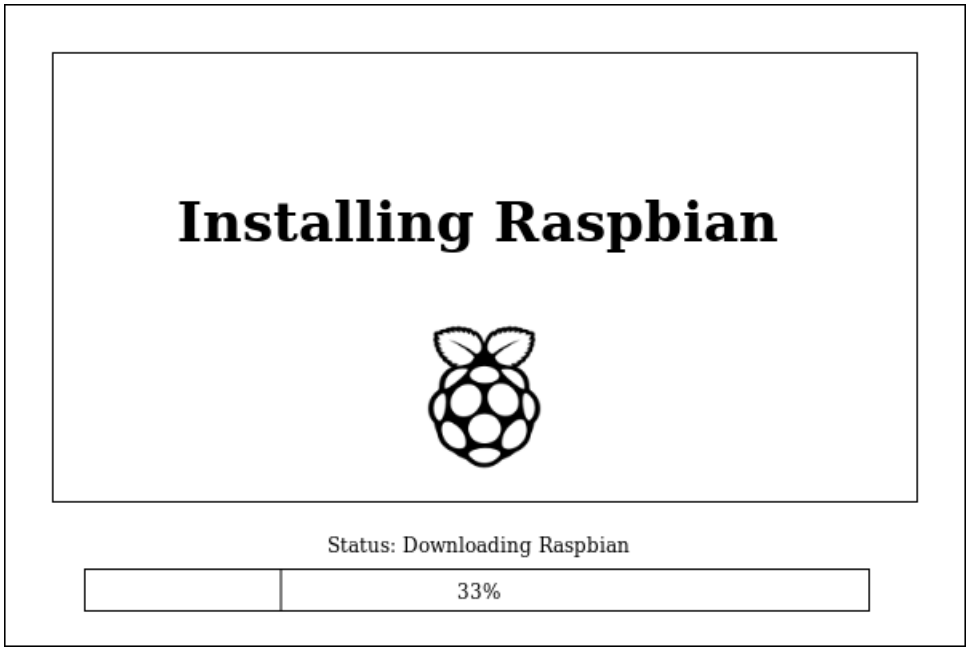


Figure 3.4: Instalation window design.

3.9 File and folder location

Several files are vital for the development of this dissertation. They may contain information for configurations or data that either the application or the daemon may need.

With this, in the table 3.15 is shown the files and folders that the RBM application will use either for configuration or to store data.

Table 3.15: RBM files and folders

Location	Description
/settings/os/	Folder used to store the metadata files of operating systems that will be installed
/settings/os/OS_name/ os.json	File containing all information about the OS that will be installed
/settings/os/OS_name/ partition.json	File containing all information about the partitions necessary to install the OS
/settings/os/OS_name/ partition.sh	Script that will be executed after installation
/settings/os/OS_name/ marketing.tar	Compressed file with all the slides shown during OS installation
/settings/rbm.conf	File containing information about configurations necessary for the application
/settings/central_app.conf	File containing information about the Central Application
/settings/installed_os.json	File containing information about the already installed OS
/proc/cmdline	File containing information about configurations used on the start of the application
/mnt/	Folder used to mount partitions
/mnt2/	Folder used to mount partitions
/RBM_watchdog/	Folder containing all the overlays
/RBM_watchdog/ rbm_daemon	Daemon that will be injected in installed OS's
/RBM_watchdog/ rbm_script.sh	Script that will be injected for the initialization of installed OS's

On the installed operating systems, the only files and folders created are for configuration and to store data related to the daemon, watchdog or initialization scripts.

The table 3.16 shows the information of this files and folders.

Table 3.16: New Operating system files and folders

Location	Description
/var/log/rbm_daemon.log	File that stores log messages from the daemon
/usr/bin/rbm_daemon	Daemon executable
/etc/rbm_daemon.conf	File containing information for the configuration of the daemon
/init.d/rbm_script.sh	Script for the initialization of the OS

3.10 Conclusion

In this chapter, was possible to give an overview of the decisions taken into account for the development of this dissertation, making it easier to implement due to the not having to worry with the design of the system.

Here was possible to see the python version and packages chosen, how the different phases of partition were thought to be, how the graphical interface windows were designed and the daemon and initialization scripts features.

Chapter 4

Implementation

4.1 Introduction

Following what was defined in the last chapter, was possible to start implementing all the components of this dissertation.

This chapter provides an overview of how the system was developed in order to fulfil all the objectives and requirements previously established. It is divided in three section, being them the development of the application, demon and initialization scripts.

Most of this chapter is dedicated to the Remote Boot Manager application, because was what took most time and effort on being developed, and being the core of this dissertation.

In order to save time, some functionalities of the source code of NOOBS ,which is open-source, were used.

4.2 Remote Boot Manager Application

In this section will be explained how the application was developed. Starting by the Remote Boot Manager Application, firstly the graphical user interface was developed, then the associated functionalities.

Is important to remember that QT was the chosen framework to develop the graphical user interface and were used some of the libraries provided by it. This libraries have their own API's and reduced the complexity and time spent for developing this dissertation.

Before any dialog being shown three checks are made. The first one is if the user presses the SHIFT key to advance for the MainWindow instantly. The second one is if an Operating system is already installed, which causes the application to show a countdown dialog to boot it.

The third one is if the application is running for the first time, which will make it start to rebuild the partition table and set up the operating system using the initDriveThread() thread.

The initial workflow of the application is shown in image 4.1 .

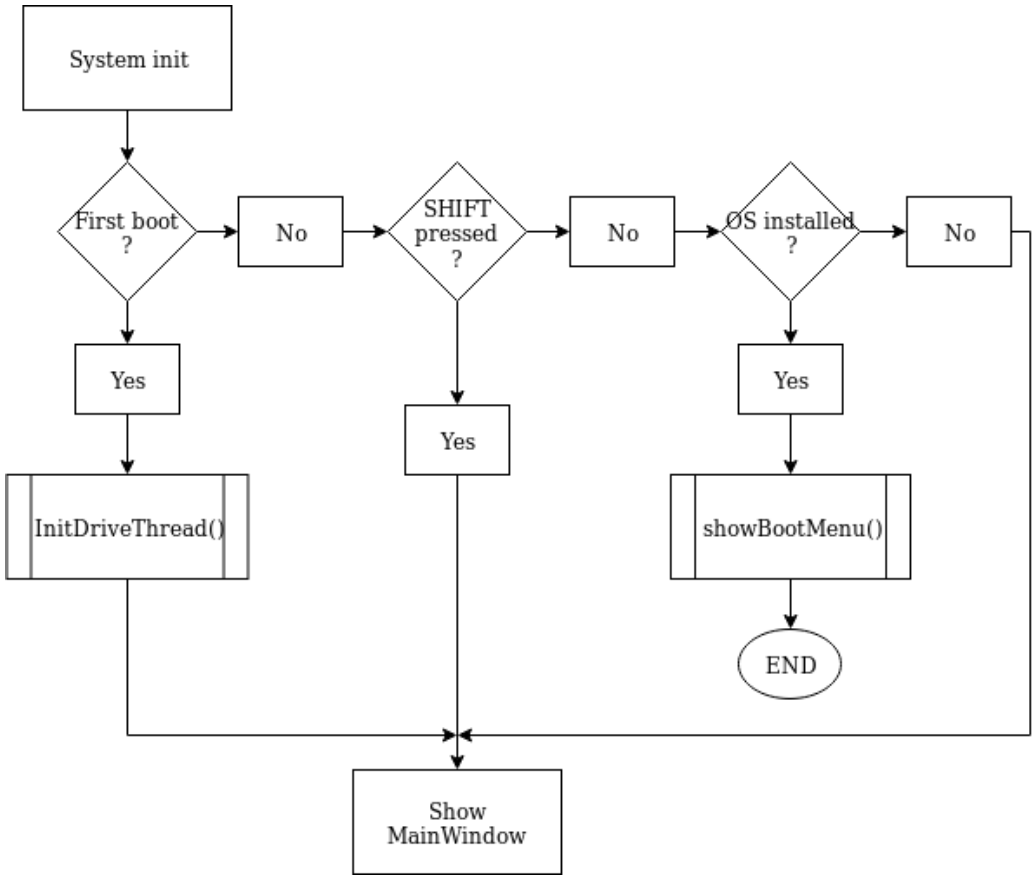


Figure 4.1: Application workflow.

If the application detects the installed operating system and the SHIFT key is not pressed the showBootMenu function will be executed. Here the system will initially perform some necessary steps before rebooting the installed operating system, like shutting down the network and unmounting the filesystem.

To check the partition where the operating system is installed, the application will read the file installed_os.json located in the settings partition. This file contains all the information about the operating systems that are already installed in the system.

After the application know which partition to reboot, a dialog with a countdown of ten seconds will appear. When this countdown expires, the system will reboot in the installed operating system by using a syscall.

To use the syscall function to reboot the system, is necessary to provide four magic values which are present in the linux manual page and are called LINUX_REBOOT_MAGIC1, LINUX _REBOOT_MAGIC2, SYS_reboot and LINUX_REBOOT_CMD_RESTART2. If this values are not provided, the system call fails and an error is thrown.

In the image 4.2 is shown the workflow of the installation method.

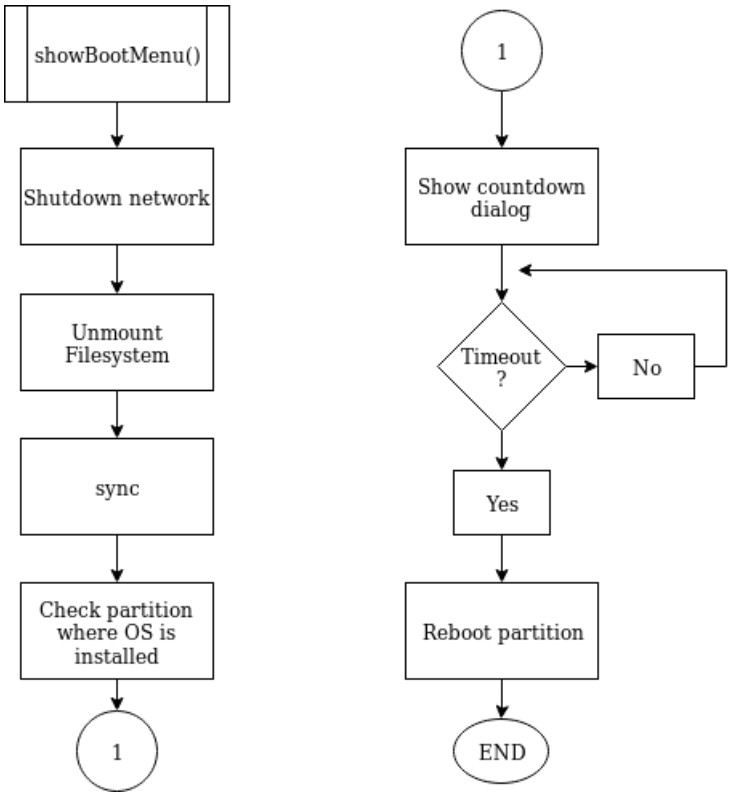


Figure 4.2: Installation workflow.

On the MainWindow initialization some checks are also made. Firstly, the application checks if the SD

card needs to be repartitioned, which happens, for example, when the application is booting for the first time. After that, the partition table is checked to guarantee that it can be read and the SD card is ready.

If the application was started because a #newimage command was received by an installed operating system, during the initialization a check will be done in the operating system logs to check for this command and the needed parameters, starting the installation of the new operating system.

After everything being ready, the settings partition is mounted and network is started.

In the settings partition are the rbm.conf file which contains configuration parameters for the GUI like the display mode and the central_app.conf which contains the information about the IP and port of the Central Application, so both files need to be read. If both the IP and port is present in the central_app.conf file, these values are saved in the GUI and the connection with the Central Application starts to being made.

After all this steps are done, the filesystem is remounted and the MainWindow dialog is shown.

In image 4.3 is the workflow of this initialization.

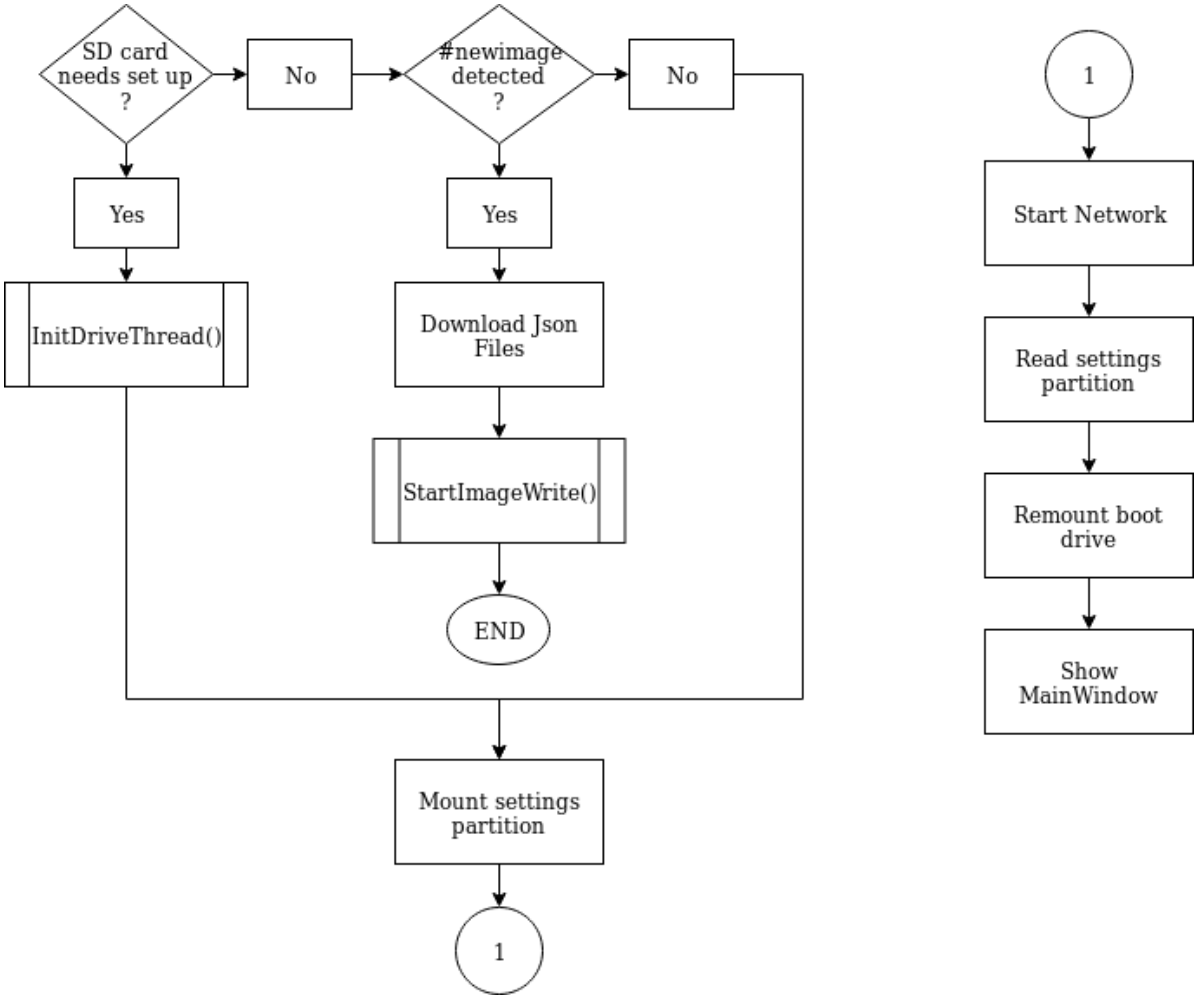


Figure 4.3: MainWindow initialization workflow.

4.2.1 TCP

For the TCP communication development, the QNetwork library from QT was used. It provides API's for connecting, sending and receiving TCP messages, making the development much easier. To keep the comunication without influencing the normal operation of the application, a thread that deals with everything related with TCP was created. When the application needs to connect, it creates a TCPThread by passing the IP address, port and a pointer to the MainWindow. This will make the thread stablish connection using the parameters that were passed and connect every signal it receives with the MainWindow, transfering the information so it can be parsed and displayed.

In the image 4.4 the workflow of this process is demonstrated.

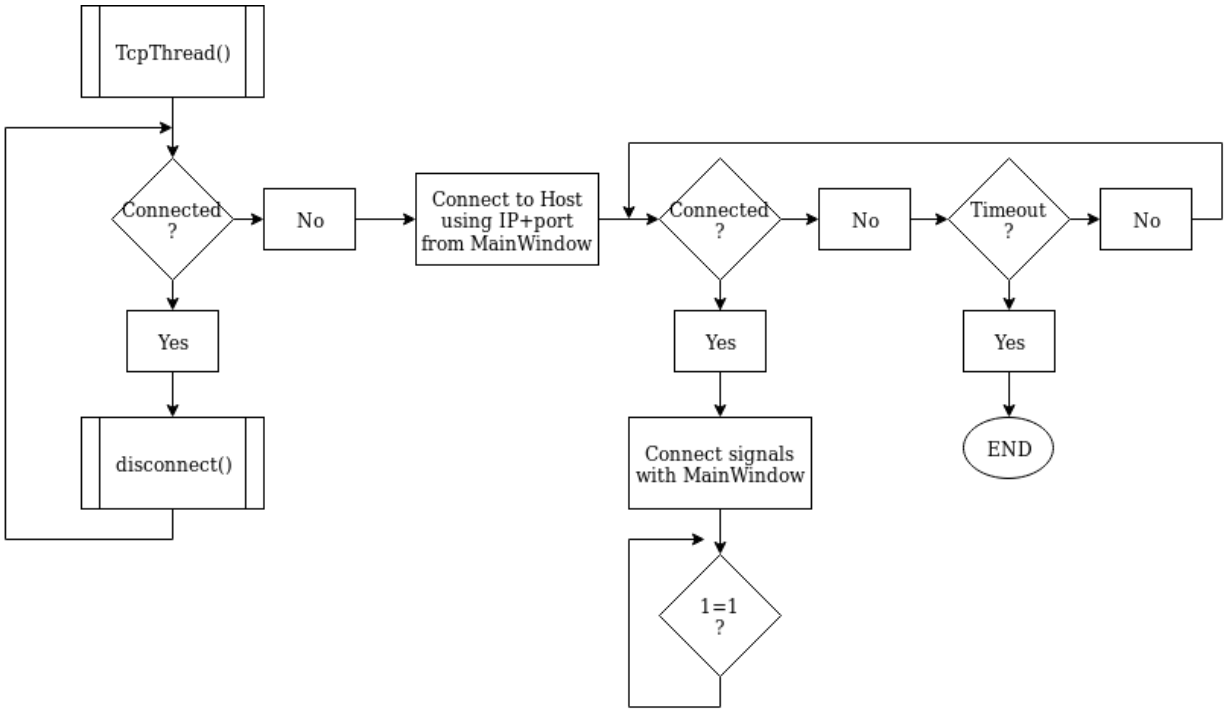


Figure 4.4: TCPThread workflow.

With this implementation the thread will try several times to connect to the host before timeout, making possible to connect to a host that is not initially ready. It also allows the user to reconnect to the same host if needed.

When the user tries to connect to a different IP or port, the previous thread is deleted and a new one is created using the new parameters. This avoid multiple connections and multiple threads at the same time.

If the application needs to send a message the previously created thread will be used. The tcpthread class provides the send_tcp() function for that purpose.

In image 4.5 is the workflow of send_tcp().

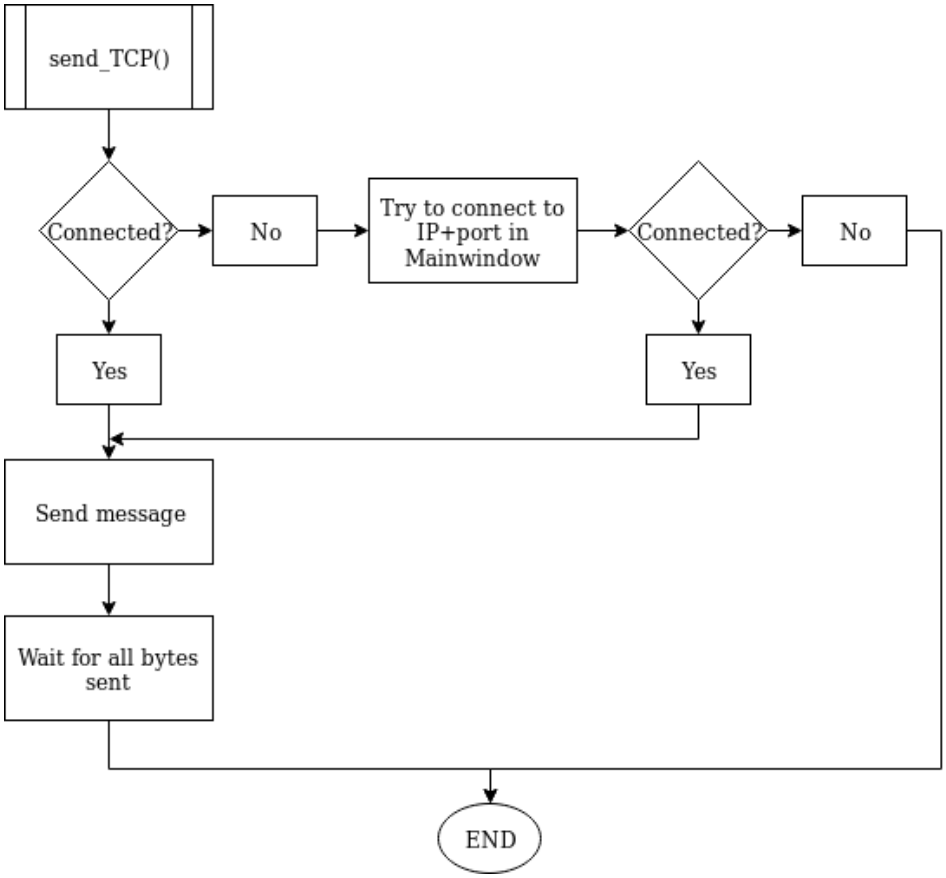


Figure 4.5: send_tcp() workflow.

To use this function is necessary to connect a signal from the MainWindow with the send_tcp() function. After this, when the signal is emitted with the message to send, the function will receive it from the signal and send it via TCP.

While the program is running the application can receive messages via TCP from the Central Application. When this occurs, the thread will receive the message, parse it and if it is valid, emit it to the MainWindow so it can be executed.

In image 4.6 is the workflow of the receive_TCP() function.

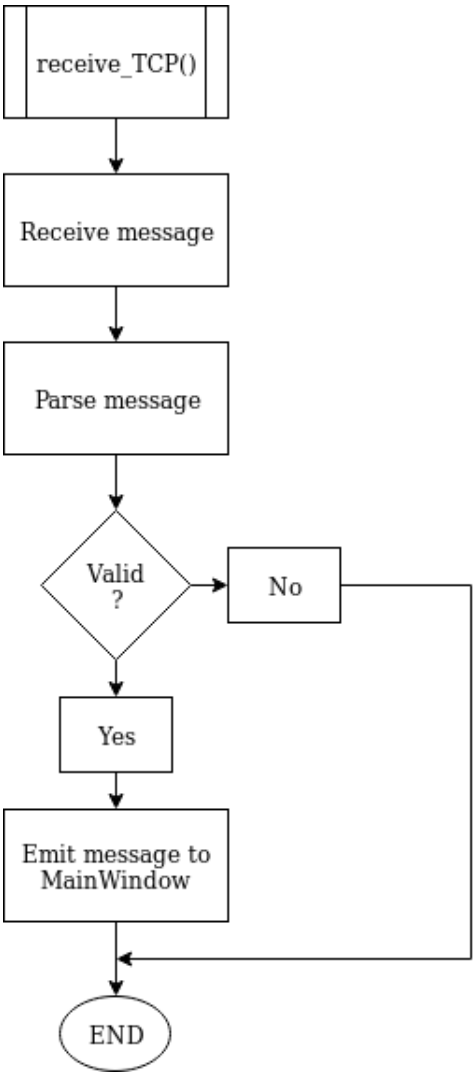


Figure 4.6: receive_tcp() workflow.

4.2.2 FTP

The library used to develop the FTP comunication was qFTP, since it provides API's for connecting, listing and change directories. which are necessary for the development of this dissertation.

Similarly to the method used to develop TCP, threads were used for the FTP connection. When the users wants to connect to a FTP server, or a #newimage command is received, a thread of the qFTP class is created.

The MainWindow can communicate with the qFTP thread by listening to signals emitted by it. This allows the application to receive information from the thread

In image 4.7 is presented the workflow of this method.

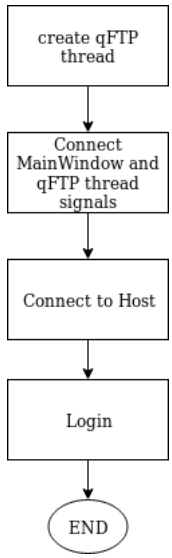


Figure 4.7: QFTP thread creation.

When the user clicks the "Download and Install" button, and a valid directory was selected, the download process of the new image will start.

First, the metadata files are download which contain the information of the OS that will be installed, then the installation process can start.

The function startImageWrite() is responsible to set up everything needed for the installation. It creates and starts the thread that does the installation process, extracts and shows the marketing images and set ups the json files with the necessary information to download the files.

In image 4.8 is the workflow of this process.

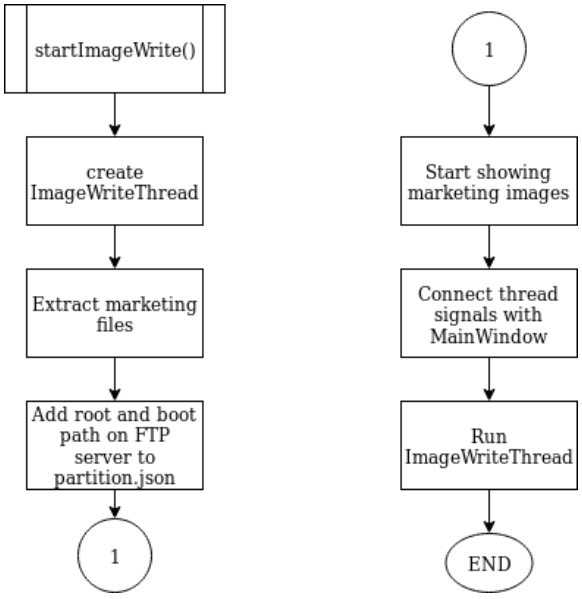


Figure 4.8: startimagewrite() workflow.

With the creation of the imageWriteThread, is possible to keep the GUI working while the OS is being installed in the background, abstracting the user from this complex procedure while keeping him informed about what is being performed.

By connecting the signals of the thread with the MainWindow is possible to show to the user the installation status. Whenever the status changes, the application also sends a message to the Central App informing about the change, using the #status command. This is possible by using the TCPThread previously created.

In image 4.9 is the workflow of ImageWriteThread().

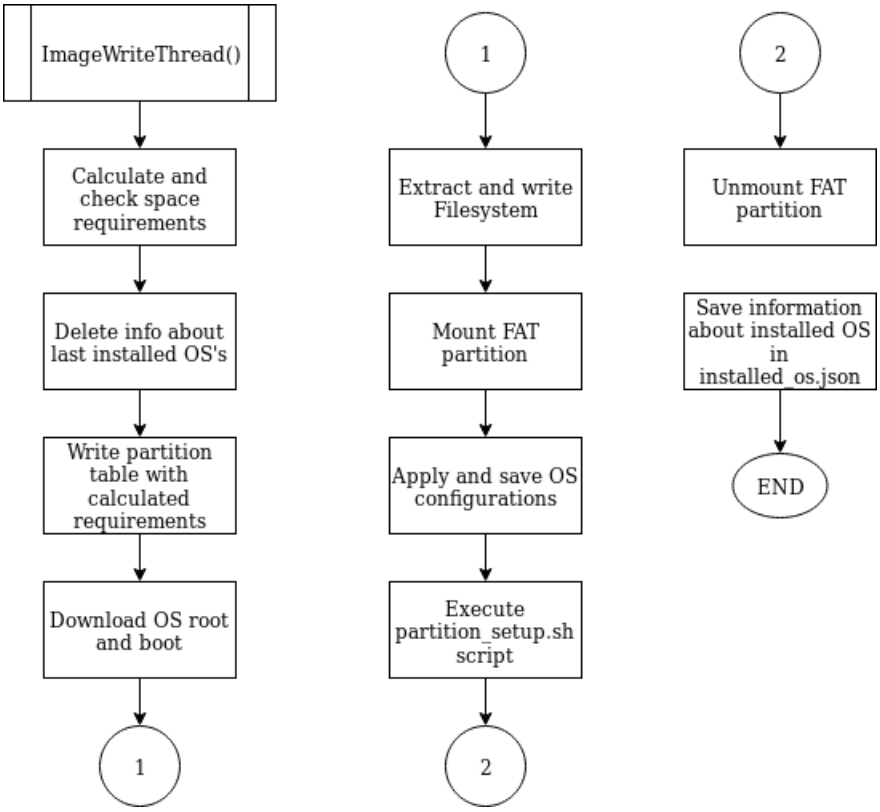


Figure 4.9: ImageWriteThread() workflow.

The first step performed by imageWriteThread is to check if there is available enough space for the OS that will be installed. This is made by checking the partition_size_nominal and the uncompressed_tarball_size values present in the partitions.json file.

If the space is enough for the installation, the thread will delete the previous installed_OS file where the information of the installed OS's is present. When a new operating system is installed all the older ones are deleted, so this file is no longer needed until the installation is finished.

Using the space requirements calculated previously, is possible to write the new partition table. This new partition table will already have included space for the boot and root partitions.

To download both the root and boot files, the wget command is used, since it has options for FTP and is commonly present in the application OS.

The OS used by the application was designed to fit only the necessary space for its normal operation, so, when the root and boot files are being downloaded, the output of the wget command is piped directly into the command that extracts and write the filesystem into the disk. With this approach, is not necessary to use any space in the disk.

When the installation finishes, the information of the new OS is saved in the installed_os.json file so that the application can boot it correctly.

4.3 Initialization Scripts

The initialization scripts were developed as shell scripts that run when the OS boots.

To accomplish this, the rc.local file is edited when the OS is installed to contain the lines .

The contents of the initialization_script.sh file are in the image 4.10.

```
#!/bin/bash

sudo -y apt-get install watchdog #install watchdog silent
sed 's+#watchdog-device+watchdog-device+g' /etc/watchdog .conf #activate watchdog
echo "#watchdog-timeout=10" /etc/watchdog.conf #add line to config
update-rc.d watchdog defaults #start watchdog on boot
sudo chmod +x /usr/bin/rbm_daemon
sudo ./usr/bin/rbm_daemon #start Daemon
```

Figure 4.10: Initialization script contents.

First, the Linux Watchdog is installed. Using the -y parameter is possible to install packages silently, avoiding checks.

After installing the watchdog, is necessary to configure it. To do this the file /etc/watchdog.conf has to be changed.

To start the Watchdog on boot, the rc.d file needs to be updated to contain it with the default configuration.

Lastly, is given to the daemon permissions to run and it is executed.

4.4 Daemon

Using daemons is a good way to provide the necessary connectivity with the Central Application without the user interference. By running in the background and providing only configuration files, the daemon can run without the user even noticing it is there.

The created Daemon will have the traditional architecture that was explained in the State of the Art chapter for the initialization. The necessary root privileges will be provided by the initialization scripts, which will run this daemon as sudo.

In image 4.11 is the workflow of the implemented daemon:

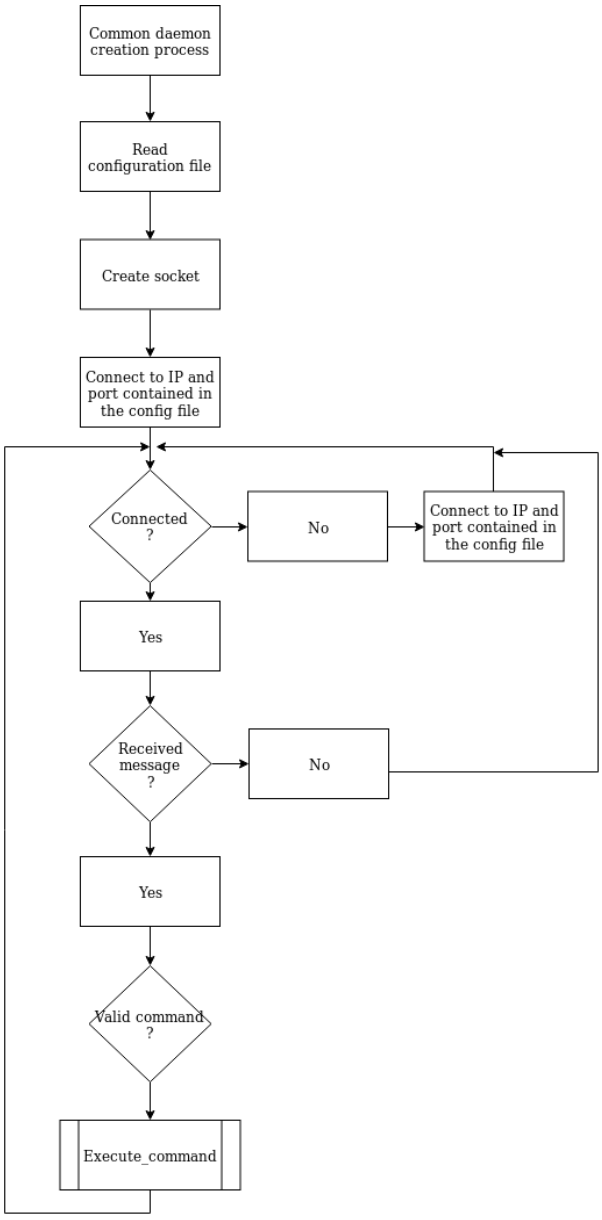


Figure 4.11: Daemon workflow.

After being demonized, the daemon will attempt to reconnect to the IP and port saved in the configuration files. During its lifetime, it will check if the connection is still established and check for received messages, executing them if valid.

By following the approach described in 4.11, is possible for the daemon to try to reconnect to the Central Application in case of disconnection, reducing the possibility of messages lost.

Is important to alert that when using the connect() API from this library, is necessary the usage of try() and catch() blocks because it throws exceptions that cause the daemon to stop if not handled.

4.5 Conclusion

In this chapter was possible to see how the system was developed, giving an insight of the API used and implemented.

This allows the reader to implement an identical system just by following what was described in this chapter.

Is important to remember that what was implemented in this chapter, follows the design made in the previous chapter.

Chapter 5

Tests and Results

5.0.1 Introduction

In this chapter is presented the tests made to the system and its results. The tests were made to the graphical user interface, the communication and the daemon.

For the testing of the graphical user interface all the buttons were tested and if they performed as expected.

The communication was tested using a terminal and checking if the messages were sent and received.

Is important to mention that each test was made at different times and some with different devices, causing the IP's to change in some screenshots.

The daemon was tested if it could run and could communicate with the Central App without problems.

5.0.2 Graphical User Interface

The first GUI to be tested was the MainWindow, here the TCP tab was tested by trying to connect to the Central App and check if the new device was added to the list.

By filling the "Server IP" and "Server Port" and pressing the button "Connect", the application should try to connect, displaying on the Central App the new device as "Connected".

The result of this test can be found in image 5.1.

IP	MAC Address	OS	Connection Status	Booting Status
10.42.0.134	B8:27:EB: 89:EF:E8B8:27:EB:DC:B...	RBM_Bootloader	Disconnected	Unknown
10.42.0.60	b8:27:eb:cb:f8:2f	Linux	Disconnected	Unknown
192.168.1.182	B8:27:EB: 89:EF:E8B8:27:EB:DC:B...	RBM_Bootloader	Connected	Raspbian: Extracting filesystem

Figure 5.1: Central App connected device.

In 5.1 is possible to observe the "Connection Status" of the selected device as "Connected", while the other devices are still "Disconnected".

Similarly, the TCP tab was tested by filling "IP/Hostname", "Username" and "Password and pressing the "Connect" button. The application tries to connect to the FTP server, displaying its contents if the connection is successfull.

In image 5.2 is possible to observe the result of this test.

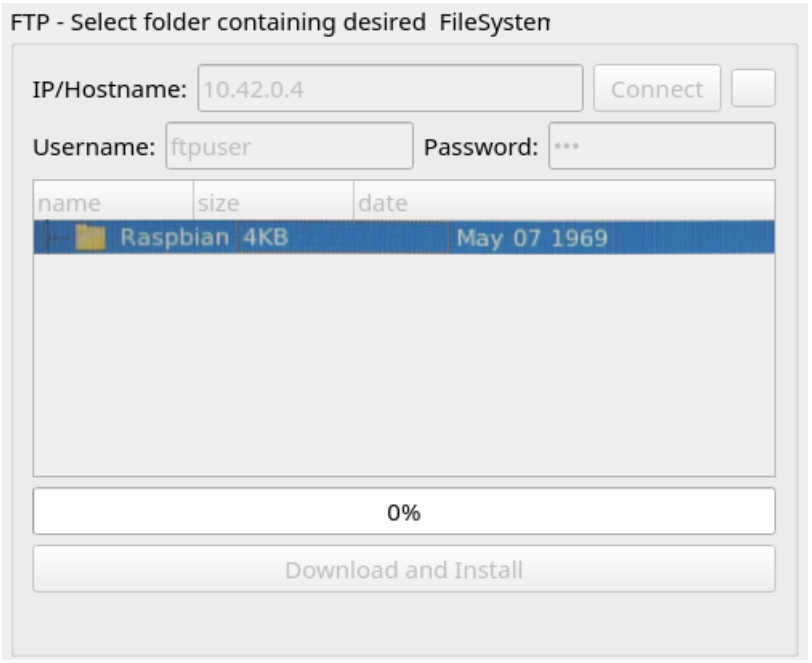


Figure 5.2: FTP test.

When selecting an OS in the FTP tab, which in this test case is "Raspbian", the application should start downloading and displaying the progress of the meta files download.

In image 5.3 is the result of this feature test.

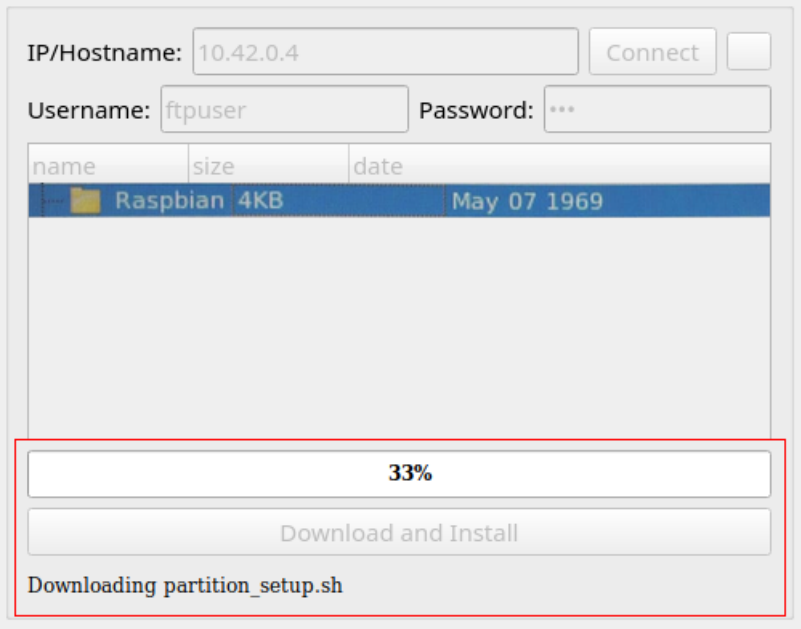


Figure 5.3: Download and install test.

Before starting downloading the meta files, a warning message is shown to warn the user that the installation of new operating systems will erase all the previous ones.

As the application downloads the meta files, the percentage of conclusion should increase and the information bar showing what file is being downloaded, change accordingly.

When the user double clicks a folder, the application should change its current directory to that folder and show its contents. If instead of a folder, the user double clicks a file, then nothing should happen.

Since the result of this tests were as expected, is possible to mark the MainWindow as tested and advance for other GUI's.

The final result of the MainWindow is presented in image 5.4.

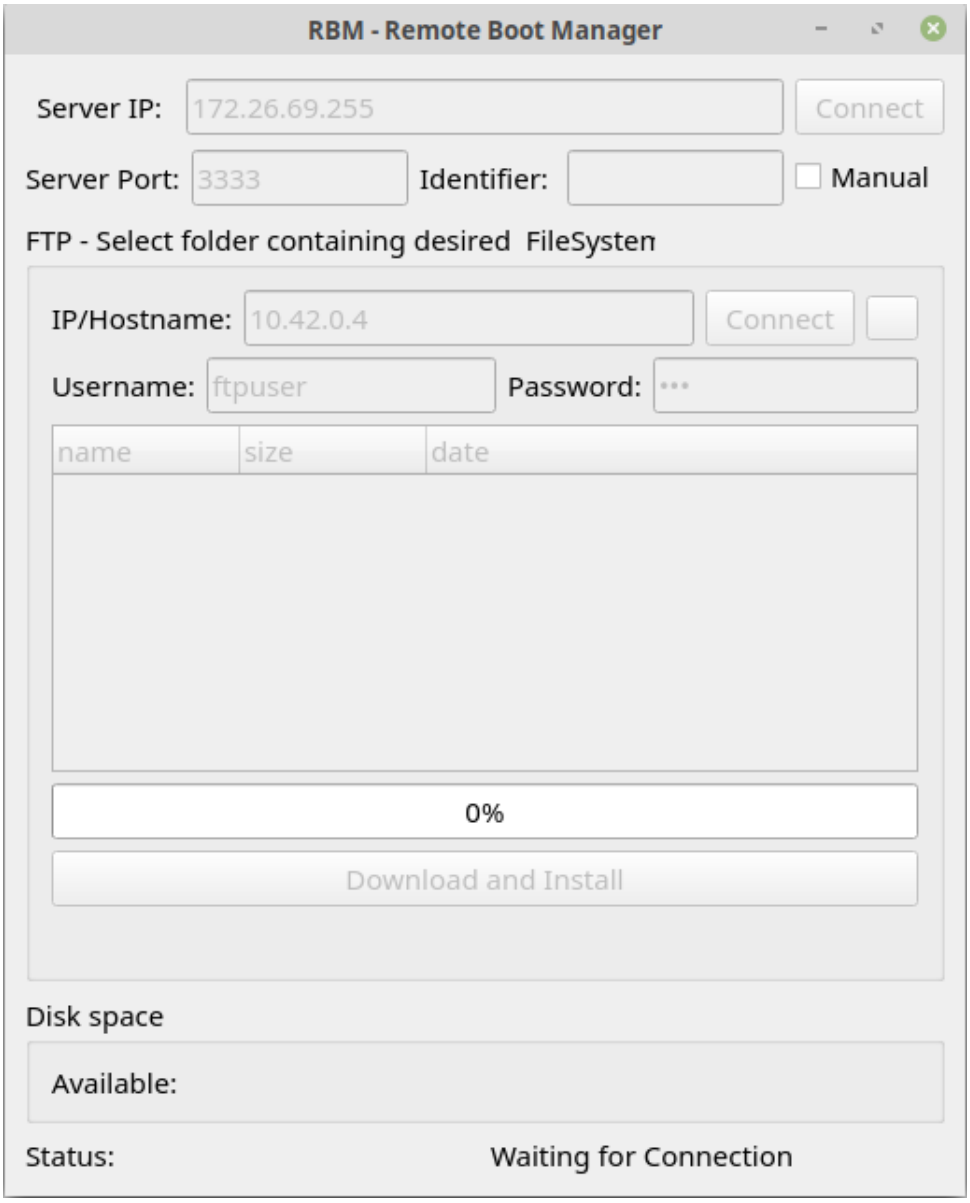


Figure 5.4: MainWindow final result.

When all the files are downloaded, a new window will show called ProgressSlideShow. This window should display images about the operating system that is being installed as well as the status of the installation.

As the instalation advances a progress bar showing the percentage of the instalation process that has been completed should increase as well as the status bar should change according to the installation status.

5.0.3 Communication

To test the communication every command that was designed in the communication language was sent and the reactions were pointed to check if they match the expected.

The table 5.1 shows the command and each of the responses received.

Table 5.1: Commands expected output

Command	Direction	Expected action	Real action
#command	Central Application-RPI	#commandoutput response	#commandoutput response sent
#commandoutput	Central Application-RPI	Display message in Central Application terminal	Message displayed in Central Application terminal
#status	RPI-Central Application	Display message in Central Application terminal	Message displayed in Central Application terminal
#getstatus	Central Application-RPI	#status response	#status response sent
#newimage	Central Application-RPI	Start installing new operating system	New operating system started being installed
#getinfo	Central Application-RPI	#info response	#info response sent
#info	RPI-Central Application-	Display message in Central Application terminal	Message displayed in Central Application terminal

To test most of the commands and if the system a complete cycle of connecting and installing a new operating system was performed. Checking the Central Application terminal was possible to verify is the messages were being received as expected and if the system was behaving well.

Image 5.5 shows the output of the Central Application after one cycle.

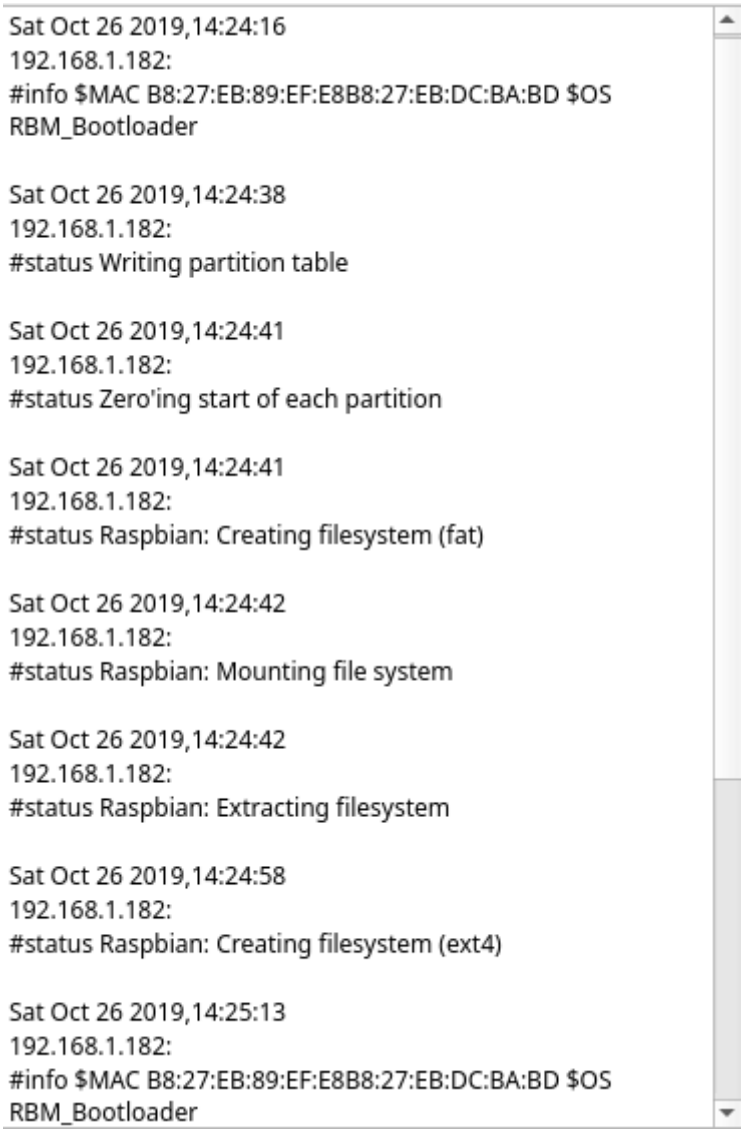


Figure 5.5: Terminal output.

The first message corresponds to the connection of the Raspberry Pi to the Central Application. When the connection was established, a #newimage command was sent to install a Raspbian Operating System which caused the application to start the installation as expected. The last #info command was received because of the Central App feature which sends a #info command to check each device status periodically.

During the installation, is possible to observe the status changes and the notification messages sent by the Raspberry Pi.

Following the messages flow, is possible to conclude that the order was correct and as expected.

5.0.4 Daemon

To test if the daemon was working properly, it was executed via terminal. Using pkill, is possible to check all the running processes, which in this case, allows to check if the daemon is running too.

In image 5.6 is possible to check the daemon process running.

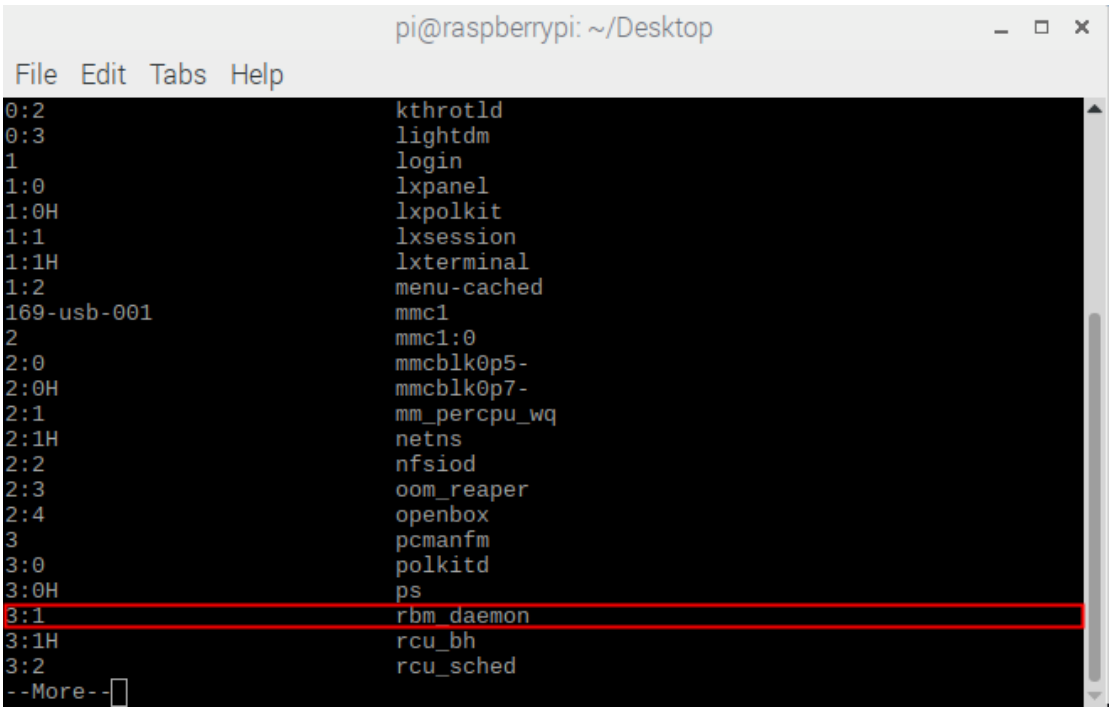


Figure 5.6: Daemon running test.

Although being tested that the daemon is in fact running, is not possible to conclude yet that it is doing what is expected which is connecting to the Central Application. To check this, the Central Application was executed and the config file of the daemon was edited so that it could connect to the IP and port correspondent to the Central Application.

Some seconds after the daemon execution is possible to verify a new connection in the Central App, proving that in fact, the daemon is running as intended.

In image 5.7 is possible to verify the daemon connection.

	Identifier	IP	MAC Address	OS	Connection Status	Booting Status
1	RBM_RPI	10.42.0.134	B8:27:EB: 89:EF:E8B8:27:EB:DC:B...	RBM_Bootloader	Disconnected	Unknown
2	RBM_RPI	10.42.0.60	b8:27:eb:cb:f8:2f	Linux	Disconnected	Unknown
3	nfo $MAC B8:27:EB: 89:EF:E8B8:27:EB:DC:B... $OS RBM_Bootloader	192.168.1.182	B8:27:EB: 89:EF:E8B8:27:EB:DC:B...	RBM_Bootloader	Disconnected	Unknown
4	RPi4	10.42.0.151	b8:27:eb:75:e3:2f	Linux	Connected	

Figure 5.7: Daemon connection test.

5.1 Conclusion

In this chapter was possible to understand how the tests were conducted to check if every component of the system was working properly.

It was also possible to check the final results of what was developed in this dissertation. By the results of the tests, the final conclusion is that everything worked as intended.

Chapter 6

Conclusions and Future Work

In this chapter are presented the conclusion and the future work that will be developed regarding this dissertation.

The conclusions were taken regarding the results of the tests realized and the accomplished objectives.

Since this is a project that will have updates and upgrades, by the time the reader is reading this document, it may be outdated.

As part of the future work, some ideas that could not be developed may be implemented and added to this document, resulting in new versions of it.

6.1 Conclusion

In the digital age we are living, information can be found in every corner, and connectivity is what allows information to reach us [31] . On each end of a connection lies a device, which in some cases can be hundred of kilometres away performing any kind of action, since the most complex to the simplest one.

Due to this connectivity is possible for each device to communicate, allowing to send messages and receive information. Even though this allows major things, it can also be used to perform unwanted actions. If the systems are not well implemented and lack security, anyone can intercept this messages, modify and read them.

Besides the success of the tests and that the system is working as intended, security will always need to be updated to be protected against current vulnerabilities and attacks, making it necessary to be constantly improved.

Is important to state that security was not a big constraint taken into in this dissertation even though its necessity and importance. It will be later developed and updated depending on the circumstances.

The approaches developed in this dissertation were taken based on the necessity of the use cases, this means that in the future some features may be improved or even completely changed to fit the necessities. The design and architecture performed as expected but the user experience can be improved with newer technologies.

Is also important to remember that the device used to develop this system was the Raspberry Pi 3 B+, so, even though the applications may work on other devices it was optimized for this specific one. Since the Raspberry Pi is still an active project, new versions of the device may be released, involving updates to what has been developed and to this document.

Performing a full cycle of burning the SD card with the RBM image, booting it, connecting to the Central App, connecting to the FTP server, selecting and installing an image, booting to the new operating system, sending a message to install a new image, reboot into the operating system installer again worked as intended, allowing it to be used in systems that need to be running for long periods of time, as was initially intended.

The option of doing every action headlessly was always a big concern, specially because was what could differentiate the RBM from other operating system installers. The usage of configuration files in the settings partition proved to be an efficient and easy way of implementing this feature, allowing the user to easily change the IP and port of application to connect to. Even though efficient, this method proved not to be the most user friendly one. With VNC, is possible to share the graphical image of the Raspberry Pi and control it. Although being a more complex method and that requires more packages and space, it gives the user the opportunity to see what is happening in the device.

The use of the Linux Watchdog, proved to be effective on preventing system and kernel crashes. In every test made to crash the system, it was rebooted every time after the timeout, allowing the system to re-enter the operating system installer application and continue the operation.

6.2 Future Work

While searching for the State of Art and designing the system, some ideas aroused interest. Unfortunately the could not be implemented, specially due to the time constraints.

To improve the way that the Central App can control the Raspberry Pi VNC can be used. With it, the Raspberry Pi can share the screen, allowing the user to see and control the system as working directly in the Raspberry Pi. This will improve vastly the user interface, but require changes to both the Central App as well to the application.

This option was already developed in PINN, so it could be a great starting point for the implementation of this feature into this dissertation.

A very important point that was not sufficiently developed was security. Despite having always concerned about it, a great improvement as yet to be made.

In the future, hardware and software to improve the security levels will be implemented. The first step will be the creation and implementation of firewalls for control access. This will allow to avoid unwanted accesses to the devices, and to create a more secure network.

The communication protocols also need to be improved, either by implementing a new one or by using versions with a better security. This protocol needs encryption to codify the messages, avoiding external users to intercept and read them, and a way to prove the reliability of both endpoints to avoid man-in-the-middle attacks.

The communication method can also be improved, depending on the use case given to this dissertation. An intranet can be created to assure to isolate the devices from the external. This might also require a change to the way the devices are connected and to the application.

In the future,Buildroot and many packages will suffer updates, so is necessary to transit all the configrations used in this dissertation to the new versions and assure that everything works as intended.

By the time the reader is reading this dissertation, many of the libraries that were used may be outdated, so they will also need to be updated.

References

[1] Freyja. *How much could software errors be costing your company?* Mar. 22, 2017. URL: https://raygun.com/blog/cost-of-software-errors/.

[2] Statista Research Department. *Internet of Things - number of connected devices worldwide 2015-2025*. Aug. 9, 2019. URL: https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/.

[3] Nick G. *How Many IoT Devices Are There?* Feb. 9, 2019. URL: https://techjury.net/blog/how-many-iot-devices-are-there/.

[4] Alara Basul. *Gartner forecats increase in IoT market by 2020*. Aug. 19, 2019. URL: https://www.uktech.news/news/gartner-forecats-20-increase-in-iot-market-by-2020-20190829.

[5] *Usage*. July 22, 2019. URL: https://www.raspberrypi.org/documentation/usage/.

[6] Liz Upton. *Introducing the New Out Of Box Software (NOOBS)*. June 3, 2013. URL: https://www.raspberrypi.org/blog/introducing-noobs/.

[7] lurch. *NOOBS (New Out Of Box Software) - An easy Operating System install manager for the Raspberry Pi*. May 13, 2013. URL: https://github.com/raspberrypi/noobs.

[8] Raspberry Pi Foundation. *NOOBS*. May 13, 2013. URL: https://www.raspberrypi.org/documentation/installation/noobs.md.

[9] procount. *NOOBS partitioning explained*. Mar. 26, 2016. URL: https://github.com/raspberrypi/noobs/wiki/NOOBS-partitioning-explained.

[10] Raspberry Pi Foundation. *How to get and install NOOBS*. Mar. 20, 2013. URL: https://www.raspberrypi.org/help/noobs-setup/2/.

[11] procount. *An enhanced Operating System installer for the Raspberry Pi*. Nov. 20, 2013. URL: https://github.com/procount/pinn.

[12] procount. *OSes supported by PINN*. Sept. 30, 2019. URL: https://github.com/procount/pinn/wiki/OSes-supported-by-PINN.

[13] max. *BerryBoot v2.0 - bootloader / universal operating system installer*. Oct. 5, 2019. URL: https://www.berryterminal.com/doku.php/berryboot.

[14] fruitoftheloom. *Understanding Multiboot Processes: NOOBS and BerryBoot*. Mar. 25, 2014. URL: https://www.raspberrypi.org/forums/viewtopic.php?f=63&t=96649.

[15] Pete Batard. *Rufus*. June 8, 2011. URL: https://rufus.ie/.

[16] Monkbot. *dd (Unix)*. Sept. 18, 2019. URL: https://en.wikipedia.org/wiki/Dd_(Unix).

[17] Wikipedia. *Preboot Execution Environment*. Sept. 28, 2019. URL: https://en.wikipedia.org/wiki/Preboot_Execution_Environment.

[18] Buildroot. *Buildroot*. Jan. 15, 2005. URL: https://buildroot.org/.

[19] Buildroot. *Buildroot Manual*. Jan. 15, 2005. URL: https://buildroot.org/downloads/manual/manual.html.

[20] Zak H. *Linux Kernel Watchdog Explained*. Oct. 20, 2017. URL: https://linuxhint.com/linux-kernel-watchdog-explained/.

[21] Wikipedia. *Watchdog timer*. URL: https://en.wikipedia.org/wiki/Watchdog_timer.

[22] IBM. *Transmission Control Protocol*. Oct. 29, 2019. URL: https://www.ibm.com/support/knowledgecenter/en/ssw_aix_72/network/tcpip_intro.html.

[23] Techopedia. *User Datagram Protocol*. URL: https://www.techopedia.com/definition/13460/user-datagram-protocol-udp.

[24] Randall Stewart. *Stream Control Transmission Protocol*. Sept. 18, 2019. URL: https://pdfs.semanticscholar.org/ba34/decca7214535d5a6f885aac01b939e6f540c.pdf.

[25] SSH Academy. *File Transfer Protocol*. Oct. 28, 2019. URL: https://www.ssh.com/ssh/ftp.

[26] SSH Academy. *FTPS*. Oct. 11, 2019. URL: https://www.ssh.com/ssh/ftp/ftps.

[27] SSH Academy. *SSH File Transfer Protocol*. Oct. 24, 2019. URL: https://www.ssh.com/ssh/sftp.

[28] Raspberry Pi Foundation. *Boot sequence*. July 22, 2019. URL: https://www.raspberrypi.org/documentation/hardware/raspberrypi/bootmodes/bootflow.md.

[29] The Linux Information Project. *Daemon (computing)*. Aug. 16, 2005. URL: http://www.linfo.org/daemon.html.

[30] Pyinstaller. *Pyinstaller*. URL: https://www.pyinstaller.org/.

[31] Huawei. *Connectivity Makes Anything Possible*. URL: https://www.huawei.com/en/industry-insights/technology/digital-transformation/connectivity.

[32] Adafruit. *Number of sold RPI 2018*. Dec. 21, 2018. URL: https://blog.adafruit.com/2018/12/21/23-million-raspberry-pi-computers-sold-raspberry_pi-raspberrypi/.

[33] Techopedia. *Computer cluster*. Oct. 23, 2019. URL: https://www.techopedia.com/definition/6581/computer-cluster.